FLEXIBLE ARCHITECTURE METHODS

FOR GRAPHICS PROCESSING

A Dissertation
Presented to
The Academic Faculty

by

Marcus Dutton

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in Electrical & Computer Engineering
in the
School of Electrical & Computer Engineering

Georgia Institute of Technology
May 2011

FLEXIBLE ARCHITECTURE METHODS

FOR GRAPHICS PROCESSING

Approved by:

Dr. David C. Keezer, Advisor
School of Electrical & Computer Engineering
Georgia Institute of Technology

Dr. Aaron Lanterman
School of Electrical & Computer Engineering
Georgia Institute of Technology

Dr. Madhavan Swaminathan
School of Electrical & Computer Engineering
Georgia Institute of Technology

Dr. Suresh Sitaraman
School of Mechanical Engineering
Georgia Institute of Technology

Dr. Mark Richards
School of Electrical & Computer Engineering
Georgia Institute of Technology

Date Approved: March 11, 2011

To my wife, Stephanie.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my wife, Stephanie, for allowing me the opportunity to continue on in graduate school. She has been a consistent supporter and encourager of whatever I've wanted to do in life. My three daughters, Kayla, Macy, and Brooke, have made the long days more enjoyable by keeping me entertained and laughing. My parents have also encouraged me throughout my education to always attain the goals I've set for myself.

I would like to thank Dr. David Keezer for advising and guiding me through my research and time at Georgia Tech. He has always been willing to help me despite the circumstances surrounding my status as a non-traditional graduate student. He has been an outstanding advisor and has been a tremendous asset in my growth.

I would also like to express my gratitude to Dr. Dan Koch and Dr. Paul Crilly, both of which I came to know very well during my undergraduate studies at the University of Tennessee. Dr. Crilly provided never-ending encouragement and wit to help me get through tough times, and he remains an excellent role model in academia. Dr. Koch allowed me the latitude to perform undergraduate research in visualizations and also sparked my desire to pursue further education.

Finally, I would like to thank to Tom Martin and Jim Parker for shaping my career at L-3 Communications. Both were instrumental in providing a platform for my research and have been excellent mentors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BMP	Bitmap
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRT	Cathode Ray Tube
DDR	Double-Data Rate
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processing
EDRAM	Embedded Dynamic Random-Access Memory
FBO	Frame Buffer Operator
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
GDDR	Graphics Double-Data Rate (memory)
GP	Graphics Processor
GPGPU	General-Purpose computations on GPUs
GPU	Graphics Processing Unit
HDL	Hardware Description Language
LCD	Liquid Crystal Display
MA	Memory Arbiter
MFD	Multi-Function Display
NRE	Non-Recurring Engineering
OP	Output Processor
PFD	Primary Flight Display
PLL	Phase-Locked Loop
PWB	Printed Wiring Board
RAM	Random Access Memory
SC	Safety Critical
SDRAM	Synchronous Dynamic Random-Access Memory
SIF	System Interconnect Fabric
SOC	System On Chip
SRAM	Static Random-Access Memory
SVGA	Super Video Graphics Array (800 x 600 resolution)
TCL	Tool Command Language
UART	Universal Asynchronous Receiver/Transmitter
VGA	Video Graphics Array (640 x 480 resolution)
XGA	Extended Graphics Array (1024 x 768 resolution)

SUMMARY

The FPGA GPU architecture proposed in this thesis was motivated by underserved markets for graphics processing that desire flexibility, long-term device availability, scalability, certifiability, and high reliability. These markets of industrial, medical, and avionics applications often are forced to rely on the latest GPUs that were actually designed for gaming PCs or handheld consumer devices.

The architecture for the GPU in this thesis was crafted specifically for an FPGA and therefore takes advantage of its capabilities while also avoiding its limitations. Previous work did not specifically exploit the FPGA's structures and instead used FPGA implementations merely as an integration platform prior to proceeding on to a final ASIC design. The target of an FPGA for this architecture is also important because its flexibility and programmability allow the GPU's performance to be scaled or supplemented to fit unique application requirements. This tailoring of the architecture to specific requirements minimizes power consumption and device cost while still satisfying performance, certification, and device availability requirements.

To demonstrate the feasibility of the flexible FPGA GPU architectural concepts, the architecture is applied to an avionics application and analyzed to confirm satisfactory results. The architecture is further validated through the development of extensions to support more comprehensive graphics processing applications. In addition, the breadth of this research is illustrated through its applicability to general-purpose computations and more specifically, scientific visualizations.

CHAPTER 1

INTRODUCTION

The objective of this research is to provide a flexible architecture for the implementation of graphics processing that can be used in many applications. Specifically, this research utilizes recent advances in FPGA technologies as well as technology-dependent algorithmic development to implement a highly-efficient graphics processing engine. Each architectural component was developed specifically for an FPGA implementation, much unlike typical GPU architectures that are designed for the structures of an ASIC. This FPGA-based architecture provides a foundation in underserved applications that require low cost, customized features, high reliability, and long device availability.

The major markets driving development in the graphics processing industry have been PC gaming as well as portable and automotive in recent years. A typical high-end gaming workstation requires massive amounts of power and provides visually appealing functionality based on three-dimensional rendering. The resulting graphics processors produce impressively rendered images, but they also have a very limited life cycle and have more features than are required in many secondary markets. Figure 1 was compiled based on online information [1, 2] to demonstrate the sheer quantity of graphics processing unit (GPU) releases in just the past few years. These leading GPU developers, ATI (now a part of AMD) and Nvidia, are designing and releasing new architectures at a remarkable rate to keep pace with each other and the game developers.

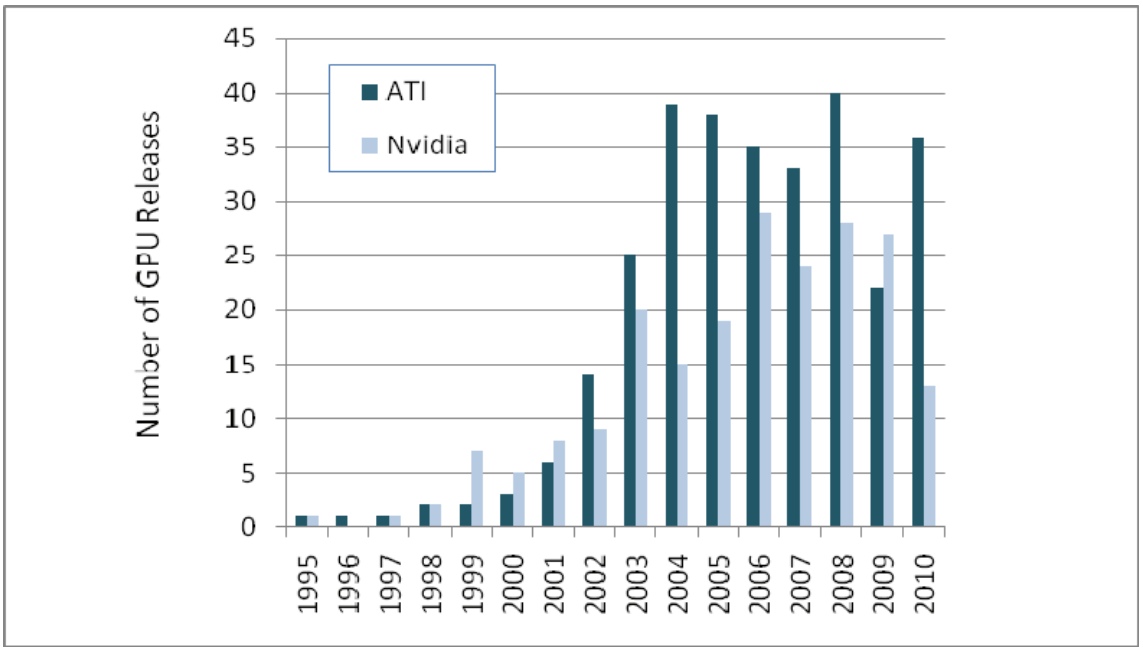


Figure 1: Compilation of released GPUs.

Figure 1 shows that in the past sixteen years, over 500 GPUs have been released by Nvidia and ATI. Once a new architecture is released, the previously released GPUs are typically not produced or even supported because the market demand has shifted. This lack of device availability and support creates a tremendous challenge for industrial, safety critical, and embedded applications that have a much longer life cycle than consumer products.

The competition between ATI and Nvidia has been motivated by a tremendous growth in the gaming market, which grew by 75% between 2000 and 2005 [3]. Since the early 1980s, the game console hardware has increased in processing frequency at a rate even exceeding the Intel processors. The increased pressures for more and more realistic gaming has resulted in considerable focus on the hardware GPU, which could be described as a special purpose accelerator that offloads much of the rendering and rasterization from the CPU.

Both advantages and disadvantages result from the reality that the gaming market drives the GPU industry. The primary disadvantage is that the GPUs are not typically suitable for embedded, low-power, or long life cycle applications. However, the main advantages are the remarkable performance and interface concepts that have developed GPUs into highly parallel computing machines. The basic motivation for such parallel architectures is simply to keep the GPU processing resources busy so that the CPU can focus primarily on the application's non-graphical requirements [4]. The software interface to the GPU is simple and does not exhibit parallelisms, but the underlying hardware is highly parallel.

As an example of a high performance GPU built specifically for the gaming market, the Xbox 360 GPU design has been well documented [5]. The design goals for the Xbox 360 architecture were flexibility, programmability, and scalability with a high-definition output resolution. To achieve those goals, the designers used three identical CPU cores that interface to a GPU core with 48 parallel arithmetic logic units (ALUs). To reduce memory bottlenecks, 700-Mhz Graphics Double Data Rate (GDDR3) memory provides a total memory bandwidth of 22.4 Gbytes/sec. The GPU, which was ATI's most recently released GPU at the time of the Xbox 360 design in 2005, runs at a clock frequency of 500 MHz. The GPU also has 10 Mbytes of embedded DRAM (EDRAM), which is DRAM that is integrated onto the same silicon with the rest of the GPU circuitry. This EDRAM is used to eliminate the depth buffering and alpha blending bandwidth requirements from the external memory interface bottleneck.

Since many of the GPUs that are released today have targeted such specific, high-volume applications, there are many other applications that are not being adequately

addressed. As an example, industrial displays and portable displays both have unique requirements that are often unfulfilled by the mainstream graphics processors. Industrial displays typically require long life cycle support, and portable displays usually require reduced functionality to minimize cost, power, and size. There is a definite need for an alternative graphics processing solution to satisfy the secondary markets, and the FPGA-based GPU demonstrated in this thesis provides the foundation for such applications.

This research provides four contributions in the area of FPGA-based processing. First, this research introduces a graphics processing architecture that was developed specifically for an FPGA. This patent-pending architecture is significant in that the overall structure as well as each module were developed to accentuate the FPGA's advantages while diminishing the FPGA's shortcomings. The architecture, which is described in Chapter 3, is highly modular to allow for each functional block to be scaled up or down according to an application's requirements. The use of an FPGA as the final target for the architectural design mitigates the risk of device obsolescence and allows the FPGA to be crafted specifically for the application requirements.

The second contribution that this research provides is a patent-pending Rasterizer architecture that uses a multi-threaded soft-core processing approach. There are many possible techniques for rasterizing lines in an FPGA, but this approach is unique in that it is highly scalable to support nearly eight times its performance with very little impact to the design. The Rasterizer architecture is also discussed in Chapter 3.

The third contribution from this research is the unique simulation approach developed to accelerate the analysis of FPGA-based processing. This approach, which is described in Chapter 4, uses a combination of standard simulation tools along with

custom developed utilities to provide a fully automated visual output based upon the FPGA design. For each simulation, the output is presented not only in a waveform but also in a text file and a visual bitmap file. The architectural research in this thesis would not have been possible without the efficiency and accuracy provided by the automated visual simulation. This same simulation approach could be leveraged for many other research areas such as video filtering or sensor fusion.

The final contribution of this research is the extendibility of the FPGA-based graphics processing architecture to support applications more broad than just basic graphics processing. Chapter 6 describes the architectural extensions for supporting comprehensive graphics processing that includes vertex processing, fragment processing, and texture management. Chapter 7 explains how this FPGA-based graphics processing architecture can also be leveraged in general-purpose computations or scientific visualizations, which combine computations with graphics processing. Together, these extensions demonstrate that this FPGA-based graphics processing architecture has a much broader impact than just basic graphics processing.

A brief background on FPGAs and graphics processing is provided in Chapter 2 to provide a foundation before the architectural thesis is presented. The two topics of FPGAs and GPUs are first addressed separately to illustrate their technology advances over the past fifteen years, and then previous works involving FPGA-based graphics processing are also discussed. The flexible architecture for FPGA-based graphics processing is thoroughly described in Chapter 3 with detailed sections on each functional module in the architecture. Chapter 4 explains the simulation methodology used during

this research and presents simulation results that were collected during the architectural research.

The fifth chapter demonstrates the applicability of the FPGA-based graphics processing architecture to the avionics industry. The requirements and options for avionics graphics processing are discussed to motivate the need for an FPGA-based graphics processor. Then the avionics implementation for the FPGA GPU is discussed with results to indicate that the architecture has satisfied all requirements. Chapter 6 provides an explanation of the architectural extensions available to support comprehensive graphics processing that would apply to a broader range of applications. Chapter 7 demonstrates the breadth of this research by showing how the architecture could be used in the adjacent application of scientific visualizations. The conclusion in Chapter 8 summarizes the key contributions of this research. Throughout this document, some details have been excluded but are available in the referenced patent applications.

CHAPTER 2

BACKGROUND

Several background topics were reviewed and analyzed in order to approach the research task of developing a flexible architecture for FPGA-based graphics processing. This chapter contains a review of the literature on both FPGA and GPU topics that aided in the research of FPGA GPU architectural concepts. First, the utility of FPGAs is demonstrated through a survey of their capabilities in both computational applications and video/image processing. Then GPU architectures and implementations are presented, including a comprehensive overview of OpenGL as well as a discussion on the use of GPUs for non-graphical applications.

Once the utility of both FPGAs and GPUs has been established, FPGA-based graphics processing is then reviewed. GPU algorithms have been the focus of intense research for several decades, but only recently have researchers also looked at maximizing the GPU efficiency in an FPGA. Typically, most of the FPGA-based GPU published research is actually just a stepping stone towards the creation of a new custom GPU ASIC. The work resulting from both the ASIC and FPGA motivation was reviewed, and the results are summarized in the concluding sections of this chapter. First the FPGA implementations of custom GPU ASICs are discussed, and then several previous GPU solutions that were actually designed for FPGAs are analyzed.

2.1 FPGA Technology

FPGAs, which are essentially fabrics of customizable logic gates, have been used for many years to implement specific functions that are not available in standard ASICs. Altera and Xilinx are the leading supplier of FPGAs, and together they captured over 85% of the market in the first half of 2010 [6]. Due to this competitive nature between Altera and Xilinx, each company continually develops new technology to take advantage of the smaller die geometries that improve cost efficiency and performance. In recent months, both companies have announced plans to offer FPGAs that are built on a 28-nm process node. Each new generation of devices reduces power and cost while increasing performance and flexibility. In addition, Altera and Xilinx continue to add more hard silicon structures such as DSP blocks, PCIe cores, and serializers/deserializers. These non-programmable features offload some of the processing burden from the programmable logic.

The continual improvement of FPGAs has allowed their use in more intensive applications such as graphics processing. The following section describes the general advantages of FPGAs and is followed by a section on the specific capabilities of FPGAs. Finally, some examples of FPGA-based image and video processing are presented.

2.1.1 Utility and Advantages of FPGAs

FPGAs can often be considered the foundation in the design of custom computing or reconfigurable computing applications. When an application requires custom hardware-based processing, an FPGA is often the first choice because of its flexibility, availability, and relatively low cost [7]. For higher volume applications, a custom ASIC

can be developed based on the initial FPGA design. However, such a customized chip requires a significant upfront financial investment.

A recent survey of reconfigurable computing architectures and design methods [7] provides an explanation of how FPGAs offer advantages over microprocessors and ASICs. General purpose microprocessors are often too expensive and not fast enough to satisfy applications that require customized processing because they can typically only process one thread at a time. Even dual-core or quad-core processors cannot compete with the speed of an FPGA-based solution that is designed appropriately [8]. In addition, the high power consumption of microprocessors (sometimes 100W or more) can exceed the power budget of entire embedded systems. ASICs offer more customized processing (often in parallel) with lower power consumption, but they suffer from expensive mask costs, which can require up to $1 million of upfront financial investment. Thus, FPGAs provide a balanced compromise between the high-power microprocessors and the low-power but expensive ASICs by offering customizable processing that can be targeted directly at the application requirements.

As an example of the utility of FPGAs, researchers at McGill University implemented a particle graphics simulator for modeling fire, smoke, and water movement in 2005 [9]. Because many of the applications for fire, smoke, and water are interactive video games, the particle system must be rendered within real-time constraints. Using a software-based implementation would not be a viable solution because the particle system processing would have to compete with the main application for the microprocessor's resources. The authors also considered using a GPU to offload the particle system from the CPU, but it was noted that the processing would still require the

CPU's involvement, which would distract it from the main application. Hence, the logical solution is to use reconfigurable logic to attack the parallel nature of the problem and completely remove the burden of particle system processing from the CPU. The calculated results prove that the FPGA-based solution is the best choice, as the FPGA can calculate over two million particles per frame. The GPU implementation is only capable of calculating 250,000 particles per frame, and the CPU implementation can only calculate 10,000 particles per frame. The results from this research are shown in Figure 2. The FPGA design, because it was purpose-built just for calculating particles and took advantage of parallel architectures, is capable of processing one particle per clock cycle. According to the authors, software methods cannot approach this metric due to the large amount of assembly code that has to be executed for each particle. Similar results could be achieved in other applications simply because the FPGA can be specifically designed to meet performance requirements.

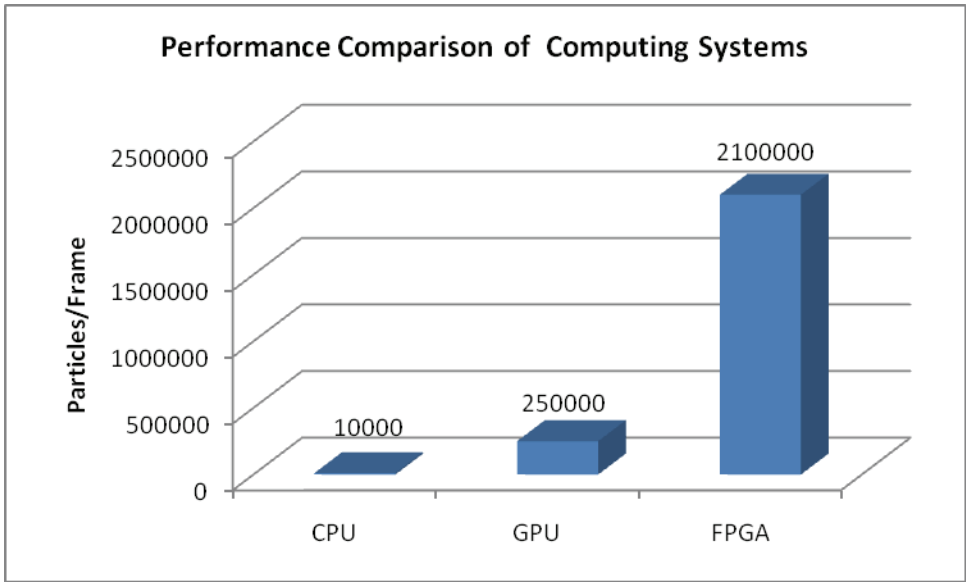


Figure 2: Comparison of particles per frame processing.

Overall, the usefulness of FPGAs lies in their inherent configurability for specific applications. This ability to customize functionality provides a tremendous advantage over fixed-function ASICs and even GPUs. However, a significant disadvantage of FPGAs is the development time required to map the algorithmic functions into an FPGA design that meets performance requirements. FPGAs can be designed with hardware description languages such as VHDL or Verilog, or they can be designed at a very high level with the use of SystemC. ASICs also require significant development time in addition to very high fabrication costs, but they typically result in higher performance designs because they do not suffer from the latency of the programmable gates found in FPGAs. However, the end result of an ASIC development is a fixed-function solution that cannot be easily modified for future applications. FPGAs, on the other hand, can be adapted continuously to address the requirements of new applications.

2.1.2 Computational Capabilities of FPGAs

Because FPGAs are general purpose programmable fabrics, they can be used for a variety of applications. In recent years, the FPGA vendors have also been including more specialized functions within the fabrics of the FPGA. For example, Altera now includes over 3,500 18x18 multiplier blocks in some of their most recently released Stratix V GS devices. The Stratix V GT devices also have four integrated 28Gbps transceivers for supporting ultra-high bandwidth applications [10]. With these and other design improvements, researchers have been able to use FPGAs for very computationally intensive tasks [11]. The data in Figure 3 was compiled based upon Altera's specifications for the Stratix FPGA family. The original device was released in 2002,

and the fifth-generation device was released in 2010. The past eight years have shown exponential growth in the number of logic elements, embedded memory bits, DSP blocks, and the speed of the transceivers.

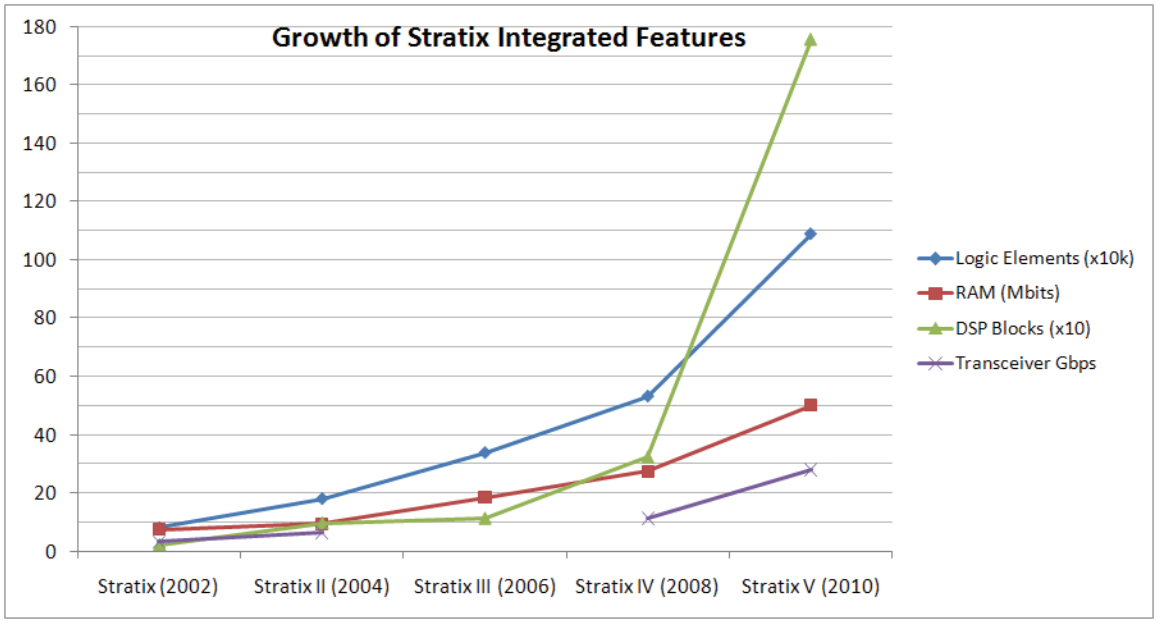


Figure 3: Growth of FPGA technology (based upon [10, 12-15]).

Matrix multiplications are commonly used in digital signal processing applications, so many researchers have tried to use FPGAs to achieve high-performance matrix multiplication. In [16], the goal was to develop an engine that is capable of producing one 3x3 matrix multiplication every clock cycle. Using Xilinx Virtex FPGAs, the researchers were able to use less than 2,500 slices (each slice has two 4-input look-up tables, two flip-flops, and some carry logic) in a design operating at 175 MHz. This means that the design can calculate 175 million 3x3 matrix multiplications per second. It should also be noted that the researcher used the original Virtex FPGA devices. Now, Virtex 5 devices are readily available that should be capable of supporting the same

design at 550 MHz. Using extrapolation, the same design could now achieve over half a billion 3x3 matrix multiplications per second. This benchmark clearly demonstrates the impressive computational capabilities of FPGAs.

FPGAs also extend their computational capabilities with the assistance of soft core processors, which are implemented using the programmable fabric of the FPGA [7]. Soft cores do not provide the same pure performance and power characteristics as hard cores because of the inherent delays through the FPGA switch fabric, but soft cores do provide flexibility. This flexibility allows developers to implement custom instructions that are accelerated in hardware. Thus, the overall performance of a soft core processor may in fact exceed a hard core processor's performance due to the joint processing between hardware and software.

2.1.3 FPGA-based Video and Image Processing

FPGAs have been used for video and image processing for many years because of their custom computational capabilities. A researcher at Siemens in India demonstrated a complete FPGA-based image processing system [17]. The author also notes that FPGAs are commonly used as co-processors in digital cameras because of their proven capabilities for parallel processing in real-time applications. There are three segments to a typical image processing system: an image sensor (camera), a processing unit (FPGA or DSP/CPU), and an output unit. In this particular research, a Xilinx Virtex 4 FPGA interfaces with a CMOS image sensor, external memory, and a VGA output. The FPGA does not actually manipulate or enhance the images in any way, but instead the FPGA

just serves as a high-speed data converter. In other applications, the FPGA design could be extended to perform edge detection, noise filtering, or contrast enhancement.

In one such example of image enhancements [18], an FPGA performs contrast enhancement on the images as they are received from the camera. In real-time (25 frames per second), the FPGA calculates a histogram of each frame's brightness level and transforms the image to have a stretched histogram at the output. Each image size can be up to 2.5 million pixels. Such real-time performance, according to the authors, is impossible with a software implementation in a traditional CPU architecture because of the general overhead required to process assembly code on a single-threaded or even multi-threaded processor. A purpose-built FPGA design, on the other hand, does not have any wasted overhead as it was designed specifically for the image enhancement application.

In addition to performing basic image enhancements, FPGAs have also been used to perform detailed image processing and recognition. Researchers at NeuriCam S.p.A and the University of Kent at Canterbury used an FPGA to relieve the host processor of burdensome parallel processing tasks while performing license plate recognition [19]. The FPGA runs at a 50 MHz clock frequency and achieves a throughput of 125 frames per second. The software implementation of this same algorithm has a clock rate five times faster but only achieves a throughput of 50 frames per second – just 40% of the FPGA's performance at a clock rate five times higher. Hence, this application demonstrates the scale of performance improvements that can be realized when converting to FPGA-based processing.

In a similar study, researchers in Japan focused on calculating the character size on the license plate instead of actually recognizing the characters [20]. Once the character size is computed by the FPGA, it can be used to determine if a vehicle is following too closely. This application requires highly efficient processing because of the real-time implications of any latency-induced errors. The resulting prototype achieved over four times the performance of a software-based implementation running on a Pentium III computer. The authors of this research also took an interesting approach to design the hardware-based algorithm. By using Handel-C, which is a higher-level language than typical hardware description language (HDL), the authors were able to circumvent the awkwardness of describing a complex algorithm in a low-level HDL.

FPGAs have also been utilized in real-time medical image processing [21] and registration, which is the process of correlating several images relative to each other. To achieve satisfactory results, image registration requires a level of intelligent algorithms that is much more powerful than typical video processing. The researchers collaborated from Cisco, University of Maryland, and Texas Instruments to consider several parallel-processing platforms, even including GPUs. As the authors noted, the GPUs have been increasing in computational throughput faster than CPUs and have therefore become useful for non-graphical applications. However, the GPU has fixed input and output interfaces as well as rigid memory restrictions. Thus, using an FPGA again proves to be the more flexible approach.

2.2 GPU Architectures and Implementations

Graphics processing has evolved dramatically over the past several decades due to both market pressures and technological advances. The evolution of the GPU architecture has been chronicled in a paper by David Blythe [22]. In the 1960s, computer-aided design and flight simulators employed primitive GPUs that were only capable of driving a vector display to stroke wireframe objects. This stroke-based rendering is best illustrated with the original Asteroids video game, which is shown in Figure 4. The overall content on the display had to remain minimal in order to allow time for the outline of each primitive to be stroked.

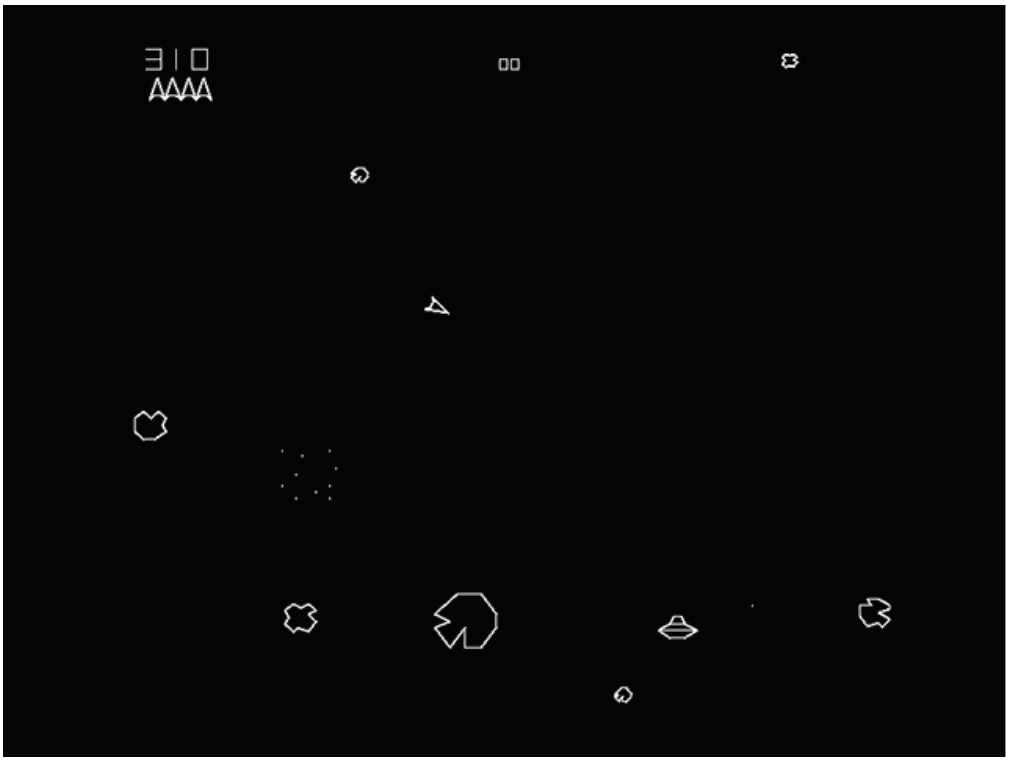


Figure 4: Asteroids – example of wireframe stroked objects [23].

In the 1970s, the development of semiconductor memory allowed the progression into raster displays, which discretely assign pixels across the X and Y axes of a monitor [24]. This move into rasterization prompted much research into dedicated fixed-function hardware accelerators. In the 1980s, IBM developed add-in raster graphics cards for PCs that were termed "video cards" because of the capabilities that they added. The game console market was also born in the 1980s with the initial Atari and Nintendo systems, and these systems used sprites to copy small two-dimensional bitmaps to the display [25]. Pac-man, shown in Figure 5, is the video game that best illustrates the concept of sprites to create motion in the display. Due to memory constraints, each of the four ghost sprites is only rendered once every four frames – but the slow response time of CRT monitors actually produced a flickering effect [26].

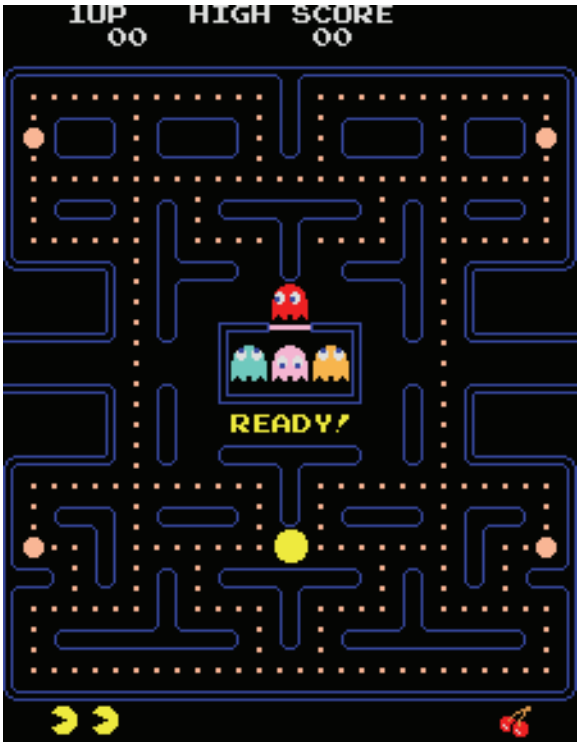


Figure 5: Pac-man – example of raster graphics with sprites [26].

In the 1990s, the popularity of the internet and other consumer applications drove the standardization of the rendering pipelines to OpenGL and Direct3D application programming interfaces (APIs) and also drove the increase in resolution from VGA (640x480) to XGA (1024x768) [27, 28]. The graphics during the 1990s were still not entirely life-like mainly because the GPU pipeline was largely fixed function. Figure 6 shows a screenshot from Tomb Raider, which was released in 1996. The wall edges are still very blockish and the skin appears very unnatural. Most of the visual effects are created only by applying a fixed texture bitmap to a flat primitive.

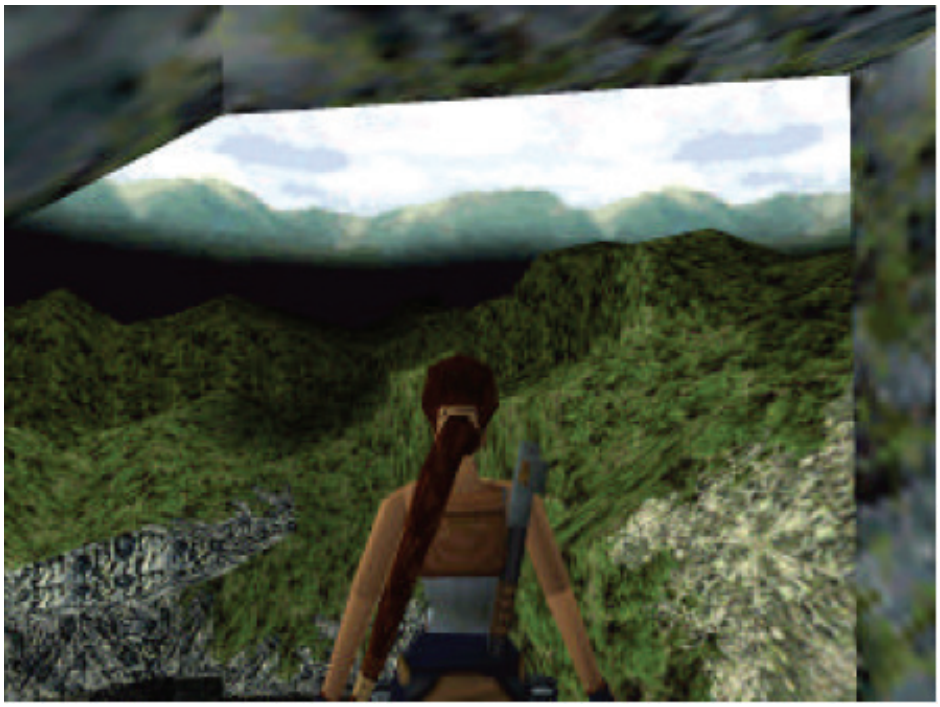


Figure 6: Tomb Raider – example of OpenGL fixed-function performance [29].

In the 2000s, the advances in the GPU market were largely driven by the gaming industry to provide more programmability. This flexibility allows a game developer to create small custom programs (called shaders) that control certain processes within the rendering engine [30]. Blythe indicates in his paper that the future presents a great

opportunity to address the gap between current capabilities and various market requirements. Because of the large investment required in each new GPU architecture, the GPUs often try to address many markets (games, multimedia, CAD, medical imaging, etc) without fully satisfying the requirements of any market. This gap between available GPUs and the actual requirements of secondary markets provides a great motivation for the proposed architectural research. This research provides a flexible platform that can be targeted to many different markets to result in exactly the desired performance.

One of those underserved markets has been the mobile and embedded graphics industry [31]. Mobile phones and other embedded devices typically have power, size, and cost constraints that dictate solutions much different from game consoles or computers. To address the downscaled graphics requirements for such embedded devices, a non-profit technology consortium (Khronos Group, Inc.) has established the OpenGL ES API to limit the far-reaching functionality of the full OpenGL API [32]. The following section provides background information on the entire OpenGL interface and is followed by sections on OpenGL implementations and research.

2.2.1 OpenGL Background

The interface between application software and graphics rendering has been standardized into the OpenGL API, which allows advanced rendering features despite a very simple programming interface [33]. There are other APIs including PostScript, X-windowing, and PHIGS [32], but OpenGL has received the most industry acceptance along with Direct3D, which is supported only in Microsoft Windows. The foundational research by Segal and Akeley presents the commonly known OpenGL rendering pipeline, which is shown in Figure 7. The first stage of the pipeline evaluates input polynomial

functions to approximate curve and surface geometry. The second stage operates on geometric primitives (points, lines, or polygons) which are described by vertices. Each vertex is transformed and lit, and then the primitives are clipped based on the viewing volume. The next stage is rasterization, in which fragments are produced that include all of the information necessary to fully describe a pixel. In the next stage, the new fragments are optionally blended with textures or modified to create a fog effect. The final stage performs a series of tests to decide if the fragment is loaded into the frame buffer to become a rendered pixel. These fragment tests include alpha blending, depth testing, scissor testing, and other criteria that may alter or remove some of the fragment's information. Alpha blending is the process of blending a new fragment with one that is already in the frame buffer. The scissor test allows the application to define only a portion of the screen that is available for rendering. All fragments that fall outside that portion are discarded.

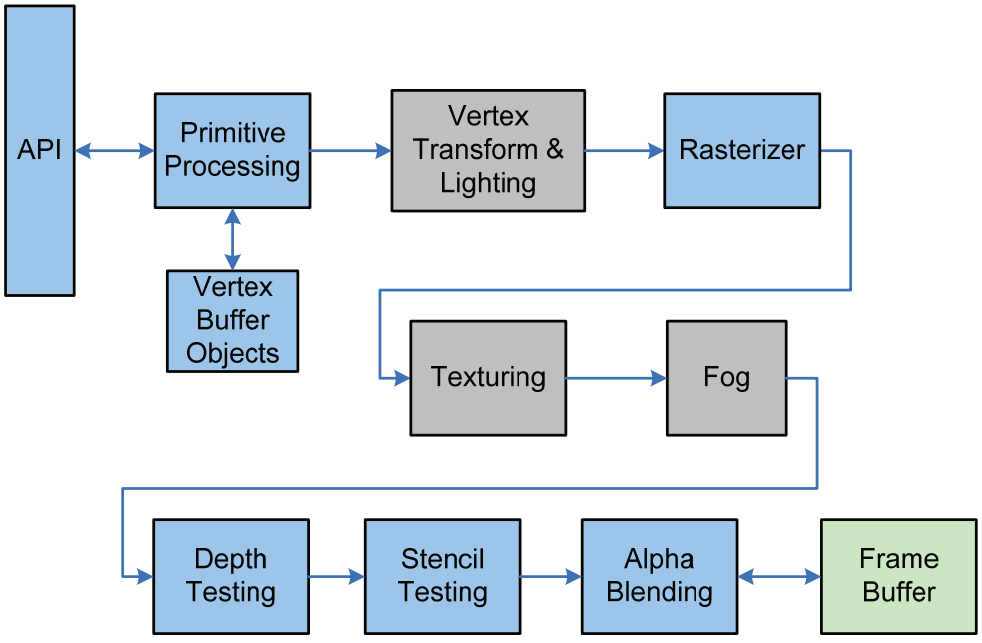


Figure 7: OpenGL fixed-function pipeline [based on 34].

As previously described, the OpenGL API has been improved upon and edited as market conditions have dictated. One such improvement has been the change introduced in OpenGL 2.0 to a programmable pipeline model, shown in Figure 8. This programmability allows the developer the opportunity to directly control certain aspects of the rendering process. These small custom programs, called shaders, are executed by the vertex and fragment processors in the GPU [35]. Shaders can now be used instead of relying on multi-pass rendering, which consumed more processing time to create novel features with literally several passes through the pipeline. Shaders can be written in the OpenGL-supported GLSL (shading language) as well as several other languages such as Nvidia's Cg or Microsoft's HLSL (high-level shading language).

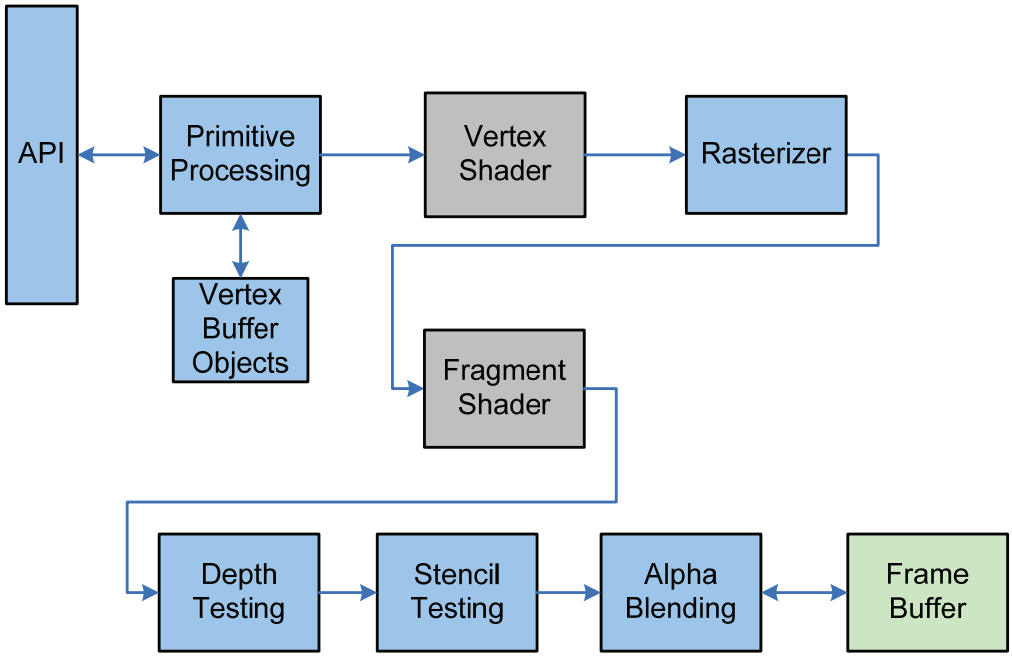


Figure 8: OpenGL programmable pipeline [based on 36].

2.2.2 OpenGL Implementations

The widespread acceptance of OpenGL has led to an abundance of OpenGL implementations used for various applications, but none have fit the requirements of a low-cost, embedded GPU that can be easily customized or extended and is obsolescence-proof. The following paragraphs describe several OpenGL implementations that have recently introduced novel architectures or ideas.

The Vincent SC Rendering Library is a complete software implementation of the safety critical (SC) subset of OpenGL [37]. The entire pipeline is written in open source software to allow developers to understand how applications are processed and to provide the opportunity to make modifications if required. Figure 9 is the output of a sample application that was written to exercise the Vincent renderer, and the code that produced this image is located in Appendix A. The performance of this software GPU is completely dependent upon the CPU, operating system, and other resources on the development system. Although the performance might not be stellar, the GPU is very convenient for quickly generating visual representations of OpenGL commands. This GPU was used quite a bit during development of this GPU architectural research to ensure that the actual results really did match the expected outcome.

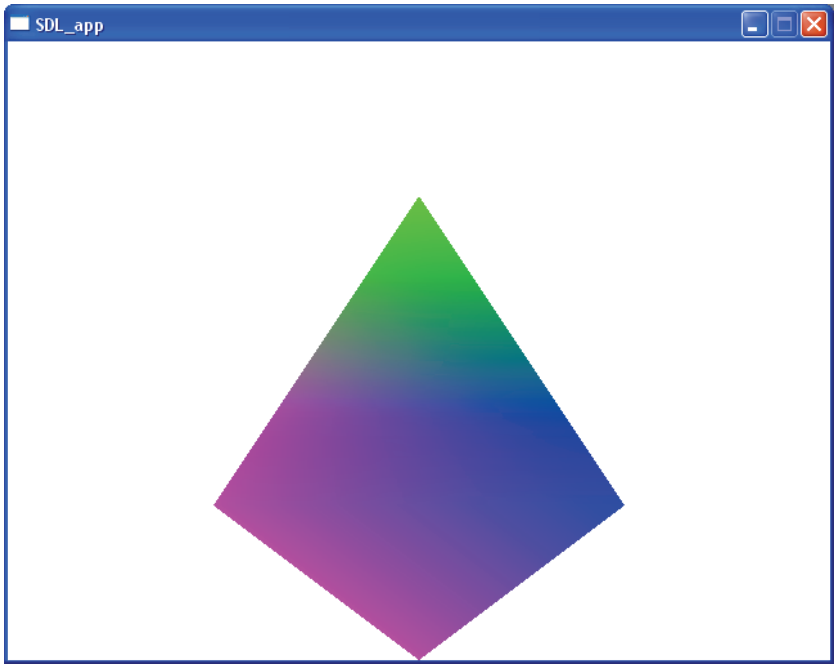


Figure 9: Sample application in Vincent Software GPU.

In a similar development, Intel has recently released details on its Larrabee GPU architecture [38] that uses multiple x86 processing cores. Larrabee is completely software-based and therefore is entirely programmable. Because of the extremely parallel architecture, there is no need for dedicated hardware circuitry for rasterization or interpolation. Instead, the authors propose that the operations can be completed in software with sufficient performance. Intel also uses software instead of fixed function logic wherever possible to maximize the programmability of the GPU. This software-based design prevents any fixed function hardware from being overdesigned to meet peak performance requirements. While Larrabee could be a powerful platform, its performance is only as good as the application written for it. Recent Intel reports have also indicated that Larrabee may not even be released as a discrete GPU [39].

Nvidia has published a paper detailing its Tesla GPU Computing Processor's architecture [40]. The Tesla architecture, originally introduced in late 2006, is a highly

parallel and scalable architecture that is notable for combining the vertex and fragment processors into a unified processor. GPUs typically process more fragments (pixels) than vertices, but the workload ratio is not consistent across various applications. By offering one type of processor for both pixels and vertices, it is not necessary to estimate in advance the application's ratio of pixel processing to vertex processing – allowing a much more versatile solution. Nvidia has actually targeted this architecture for general-purpose computations, and many of the Tesla products do not even have a video connector.

The Tesla architecture has 128 streaming-processor cores that operate at 1.5 GHz, requiring 150 W of typical power in the GeForce 8800. [40]. Even though Tesla provides a powerful platform, it does not satisfy the long-term device availability and low power requirements of embedded and industrial applications. Implementing the Tesla architecture directly in an FPGA would impact performance because of the limited clock speeds and embedded memory. The GeForce 8800 hosts 768 MB of GDDR3 memory; by comparison, the maximum on-chip memory in Altera Stratix IV devices is just over 2.5 MB [12]. However, a new GPU architecture that is crafted specifically based on the FPGA's capabilities can maximize overall performance given the FPGA's memory limitations.

Nvidia's most significant advancement since the Tesla architecture is the Fermi based GPU. Fermi was designed to meet the needs for GPU computing – not specifically for graphics applications [41]. There are three billion transistors in the massively parallel design that has 512 streaming-processor cores. Each of these cores can execute a single floating-point instruction per clock. Double precision arithmetic is now also supported

with a 4x faster performance over the previous GT200 architecture. Multiple shared memories and on-chip caches are now available as well as 384-bit memory interface that supports up to 6GB of GDDR5 DRAM.

2.2.3 Academic Research

Academic researchers have also investigated OpenGL pipeline implementations in order to better understand both algorithmic and architectural concepts. In one such effort in 2006, researchers at Seokyeong University implemented a vertex processing unit based upon the OpenGL ES 2.0 API [42]. This research was one of the very first implementations of ES 2.0, and it was completed entirely in software to allow for complete programmability. However, the performance was not optimized and the maximum frame rate was inversely proportional to the number of vertices and number of instructions in the shader. This performance drop would prevent such a software-based design from fulfilling many of the speed and rendering requirements of real-time embedded applications.

To address performance issues, researchers from Princeton and NVIDIA developed an approach in 2007 to use point-based rendering instead of the standard triangle-based approach [43]. The researchers claim that rendering with anti-aliased triangles is challenging because very small triangles can fall between sampling points and thus be eliminated in rasterization. The hardware-accelerated rasterizer for point primitives was implemented in two FPGAs that were clocked at only 70MHz. The thrust of the research was the accomplishment of anti-aliasing by using Elliptical Weighted Average (EWA) surface splatting, which is a point rendering algorithm. The final

implementation, however, required two large circuit cards and a dual-FPGA setup that does not lend itself towards mobility or low-power applications.

PixelFlow is a very powerful architecture that was developed at the University of North Carolina [44] to definitively exploit parallelism to quickly generate graphics. The actual hardware is based upon a Flow Unit which includes a dual-processor GPU, memory, and a custom interconnect ASIC that connects all of the GPUs to the Geometry Network. The architecture is completely scalable with up to 256 Flow Units. The software interface to PixelFlow is OpenGL in order to demonstrate the programmability of its processors. Although PixelFlow is highly scalable, its inherent overhead associated with the custom Geometry Network would limit its effectiveness in embedded applications.

2.3 Non-Graphical GPU Research

The wide impact of this research is best illustrated by reviewing how GPUs are not used just for graphics processing. Much existing literature demonstrates the GPU's effectiveness in a number of applications that are unrelated to actual graphics processing. An online community is dedicated to the research of general-purpose computing using graphics hardware [45]. According to the GPGPU (General-Purpose computation on GPUs) website, such research has been underway since the late 1970s, but the research has increased greatly in recent years as graphics processors have rapidly advanced.

There are a couple reasons why GPUs are becoming more attractive to researchers with large computational problems. Many traditional microprocessors are not power

efficient, and we may be approaching limitations of traditional Silicon devices with regards to lowered voltages and increased clock speeds. The GPGPU presentation at SIGGRAPH 2007 [46] explains why the GPU is unique yet versatile enough to be used in many applications. The GPUs are based on fine, lightweight threads, and the performance of one single thread is clearly inferior to traditional processors. However, GPUs gain the advantage through the parallelism of their architecture to reduce overall latency. The GPUs also have a very well established programming interface, and there is a GPU market of over 500 million per year. Thus, with such a large market and interest, the competitive pressures result in a continual effort towards further enhancements and improvements to GPUs.

Medical imaging is one of the leading users of GPUs for non-graphical applications. In [47], it is demonstrated that a GPU can be optimized for use in magnetic resonance imaging (MRI). Typically after the scans are conducted, a computer performs extensive image reconstruction algorithms to process all of the data into a meaningful format. The quality of the reconstruction algorithm directly determines the usefulness of the MRI exam. It has been determined that using an advanced least-squares (LS) algorithm is much superior to the traditional fast Fourier transform (FFT) method, but the LS algorithm requires significantly more computational horsepower. This increased computational burden must somehow be implemented on cost efficient hardware in order for the solution to be feasible in a clinical setting.

Researchers implemented the LS algorithm on an Nvidia GeForce 8800 GPU not only to take advantage of the parallel processing, but also to use some of the other features such as the constant memory and special function units. The results were

dramatic as the optimized GPU algorithm was able to finish processing over 250 times faster than the single-precision CPU algorithm. Thus, the MRI results can be reconstructed in three minutes instead of six hours, and the implementation is actually viable in a clinical setting. The authors also acknowledged that GPUs will likely be the focus for computational advances now that CPU progress has slowed due to concerns over power consumption.

Separate research has also been conducted regarding the usefulness of GPUs in accelerating a cone-beam Computed Tomography (CT) reconstruction algorithm [48]. The FDK Filtered Backprojection algorithm involves projection-space filtering, back-projection, and volume-space weighting. The authors considered several possible approaches including a GPU, an FPGA, and a processor. The GPU was ultimately chosen based on its single instruction, multiple data (SIMD) architecture and ease of programmability with the Nvidia CUDA (Compute Unified Device Architecture) interface. CUDA was developed to allow programmers the ability to write C-code to use Nvidia GPUs for any computationally intensive task, not just a graphics application. The authors compared their implementation with prior research that was completed on a CPU and an FPGA, and they showed a 4x improvement over the FPGA and a 10x improvement over the CPU. The authors note that the FPGA implementation was done several years prior and that FPGA technology may have improved enough since then. Thus, the programmability and flexibility of FPGAs seem to be a ripe area for future research with complex algorithms.

The CUDA software platform, which was released by Nvidia in 2006, has allowed researchers not familiar with GPU architectures to in fact use GPUs for powerful

parallel computations. CUDA is significant in that in hides the GPU hardware complexities to create a simple high-level programming model [49]. The application developer does not need to write threaded code. Instead, a hardware thread manager coordinates all of the multithreading that is based upon the GPU's parallel architecture.

Researchers at Georgia Tech Research Institute have recently published results after using CUDA to perform QR matrix decomposition on GPUs [50]. Using an Nvidia GeForce GTX280 graphics card, the QR implementation achieved 143 GFLOP/sec, which is nearly five times faster than a benchmark algorithm implemented on an Intel quad-core processor operating at 2.4GHz. The authors achieved their results after careful analysis of the possible QR decomposition algorithms to determine which algorithm best matched the capabilities inherent in the GPU architecture. In the QR decomposition research, the three factors used to determine the effectiveness of the algorithm were the patterns of memory accesses, synchronization requirements between different elements, and the scalability of the algorithm to exploit parallelism.

While CUDA is specific to Nvidia GPUs, the Khronos Group has been developing the initial specifications for OpenCL, which is a cross-platform standard for heterogeneous parallel programming [51]. This standard is an API that uses kernel execution as its main model of programming and allows implementations on CPUs, GPUs, DSPs, and other processors. The first release of OpenCL's specification occurred in late 2008, and since that time OpenCL 1.1 has been issued and AMD, Nvidia, ZiiLabs, VIA, and IBM have all released OpenCL drivers for their products. The generic nature of OpenCL that allows execution on a variety of hardware models also provides an

interesting perspective on FPGA-based graphics processing since both FPGAs and GPUs are involved. This perspective is further explored in Chapter 7.

2.4 FPGA Implementations of GPU ASICs

FPGAs are often used for ASIC prototyping in order to avoid multiple ASIC fabrication costs due to design changes and improvements. In fact, some FPGA devices are specifically developed so that they provide maximum logic elements with little or no hard cores such as PCIe endpoints or high-speed transceivers. For example, Altera's Stratix V E device family was optimized with over 1 million logic elements for ASIC prototype applications. The following GPU research was conducted in FPGAs but was ultimately targeted for a custom ASIC. As such, the FPGA's unique architectural elements were not taken advantage of or even utilized. However, this research is important to better understand the performance and possibilities of FPGA-based graphics processing. The following two pieces of research that are described each contributed GPU architectural concepts without taking advantage of FPGA structures.

In [52], the researchers aimed to create a 3D graphics engine that results in minimal hardware cost but is capable of generating graphics for small to medium screen sizes in commercial electronics. The final target of a custom ASIC is obviously chosen to minimize unit cost while capitalizing on the high volumes expected in consumer electronics. As shown in the high-level block diagram in Figure 10, the researchers used the Advanced Microcontroller Bus Architecture (AMBA) interface to allow an easy integration into a System-on-Chip (SoC). ARM Ltd released the AMBA specification in

1996 to standardize the way in which different modules interface in SoC designs [53], and it has arguably become a defacto standard for on-chip interconnect. The key advantage to this standardization is that there is now a large community of IP suppliers that provide modules with AMBA interfaces. This ecosystem of compatible modules promotes the AMBA interconnect fabric. AMBA's main competitors are the Wishbone bus from Opencores, the CoreConnect interface from IBM, and Altera's Avalon interface.

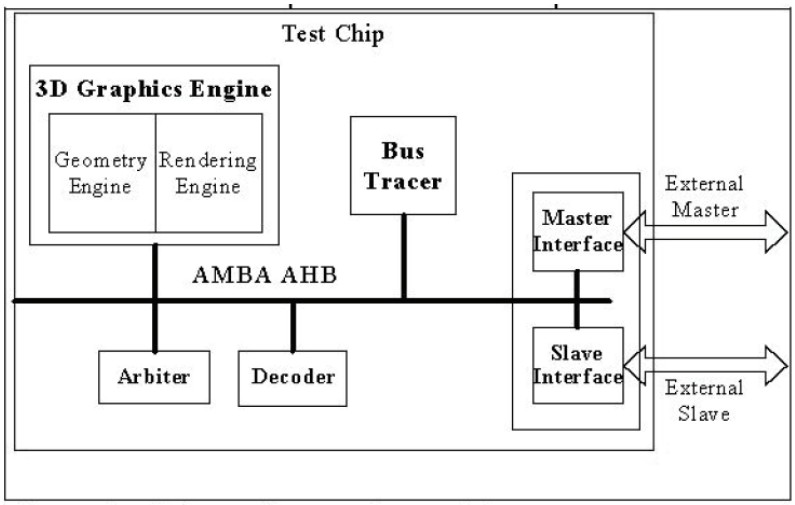


Figure 10: High-level block diagram of ASIC 3D graphics engine [52].

The 3D graphics engine is shown as a module on the AMBA interface along with an arbiter, decoder, and bus tracer. The bus tracer was included to allow diagnostic visibility into the AMBA interface. The OpenGL ES pipeline was divided into two sections – a Geometry Engine and a Rendering Engine. The Geometry Engine is responsible for all vertex transformations and requires quite a bit of mathematical complexity. As such, the researchers divide the Geometry Engine into three pipeline stages with each stage requiring 16 clock cycles. The Rendering Engine is also divided into two pipeline stages – rasterization and pre-fragment operation units. The number of

rasterizers can be increased up to a total of four per chip if higher throughput is required. The ASIC design used two rasterizers with performance that was projected to be sufficient to satisfy the requirements.

Although the performance results of the ASIC solution were not available, the FPGA implementation of this architecture runs at 139 MHz and produces 8.34 million vertices per second and 278 million pixels per second. The ASIC was scheduled for fabrication on the 0.18um process with a 16 square mm die area, and it is expected to require 400mW of power. These performance numbers should allow the GPU to be useful in relatively low-resolution commercial electronics where the focus is on minimal hardware cost.

In a second effort, the researchers focused more directly on developing a geometry engine for mobile 3D graphics [54]. This research was motivated by mobile electronics that are not served well by the mainstream GPUs. The term "geometry engine" is important in this research because the design is actually not a full GPU. Instead, the researchers chose to offload only the vertex processing portion of the standard GPU pipeline. As shown in the block diagram of their approach in Figure 11, the Rasterizer actually resides in the ARM processor as software. This research is also an SoC solution that utilizes the AMBA Advanced High-performance Bus (AHB) to maximize versatility and integration with other modules.

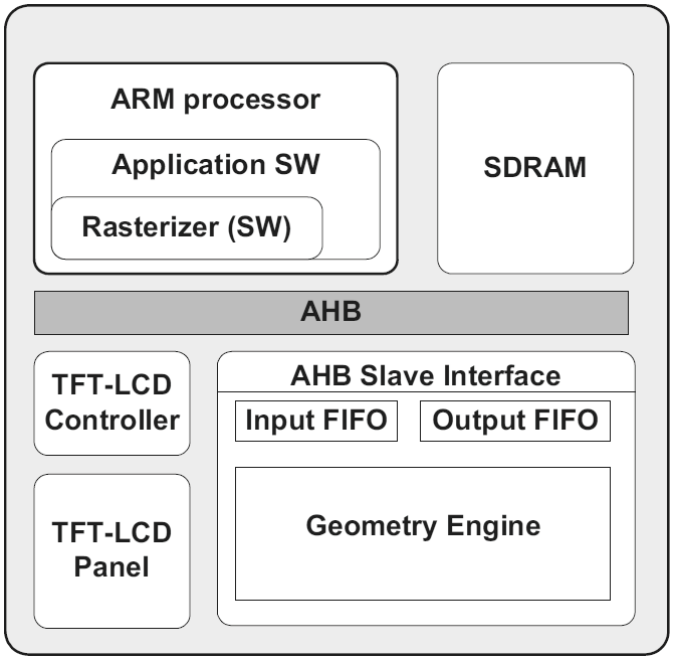


Figure 11: Block diagram of mobile electronics ASIC graphics processor [54].

The researchers divided the Geometry Engine into two sub-modules, a transformation engine and a lighting engine. The transformation engine maximizes throughput by optimizing the culling and clipping operation. In traditional architectures, the vertices are mapped to the screen coordinates after they are clipped. However, this research showed that a significant savings can be realized by simplifying the clipping process. Figure 12 shows how the researchers rearranged the sequence of events such that the clipping has now become a very simple cull and sort operation at the very end of the transformation process.

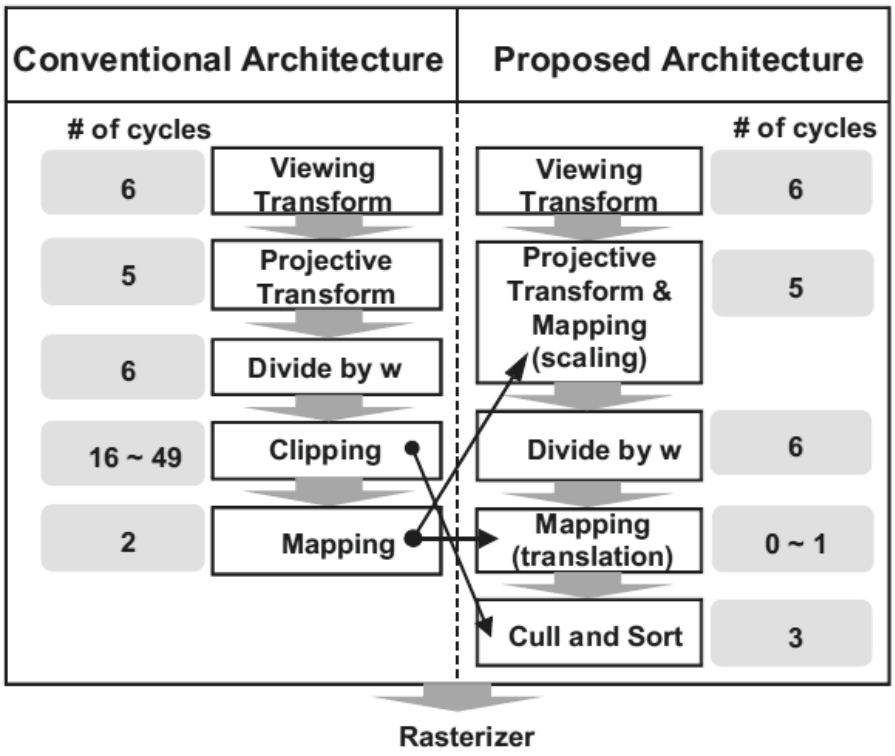


Figure 12: New architecture to optimize clipping operations [54].

The researchers designed the lighting engine after analyzing the mathematical operations required for various lighting operations. The OpenGL lighting equation allows for up to four different light types: ambient, diffuse, specular, and spot. The factors and positions of each of these lights are all taken into account to produce the final color components for each vertex.

The geometry engine was implemented in Verilog-HDL to account for up to eight light sources and was tested in an FPGA. The output of the geometry engine is fed to the Rasterizer that is located in the ARM processor. The design for just the geometry engine requires 273,600 logic gates and can process up to 16.8 million vertices per second, depending on the lighting conditions. If all eight lights are used, the throughput drops

down to approximately 3 million vertices per second. The geometry engine operates at only 100 MHz, with no explanation given as to the relatively low frequency. Perhaps the FPGA was not capable of meeting the timing constraints for higher frequency clocks.

2.5 GPUs Designed for FPGAs

2.5.1 Academic Research

Several academic researchers have recognized the utility in FPGAs as the final target for graphics processing applications. In fact, as early as the mid-1990s, some researchers began realizing that FPGAs were a promising area for GPU research [55]. This section chronologically details the results from the academic research community over the past sixteen years.

In 1994, a series of experiments was conducted to determine if FPGAs were a suitable technology for the implementation of graphical algorithms [56]. The researchers implemented three graphical algorithms: circle outline, filled circle, and shaded spheres. Both of the circle drawing implementations used the midpoint circle algorithm, with the filled circle also requiring the extra rasterization of the circle's interior. The shaded sphere implementation used a generalized fast spheres algorithm. In each case, the researcher attempted to perform as many calculations as possible in parallel, but the results showed only a 10 frames/second performance with very minimal content. Nonetheless, the research was instrumental in beginning the focus for graphics processing in FPGAs.

In 2000, research focused on the ability to customize graphical architectures for specific applications [57]. The researcher motivated the problem by describing five different applications that each has individual requirements for vertex processing, fragment processing, scan conversion, texture mapping, hidden-surface removal, and frame rate. Rather than require a GPU ASIC to perform well in all of these different applications, the researcher proposes that a flexibly-designed FPGA is more appropriate. The researchers started with a software model for the entire graphics pipeline and then partitioned the elements into software and hardware portions. The hardware portions were designed in the high-level Handel C language to allow ease of transition from the software model. The researchers focused on three features that were customized based upon the application: output formats, lighting and shading models, and texture mapping. The final implementation had an OpenGL interface with frame rates that were approximately 20% of the GPU ASIC results. In 2004, the same researcher implemented six different shaders in a Xilinx Virtex II device. The results showed that GPU ASICs still had clock speeds 3-4 times higher and fill rates 10-20 times higher [58].

In 2005, researchers attempted to implement a complete FPGA-based 3D graphics system [59]. The research showed that developing such a system is not trivial and requires quite a bit of ingenuity. Figure 13 shows the system overview, which includes a CPU, memory controller, VGA controller, and the 3D pipeline.

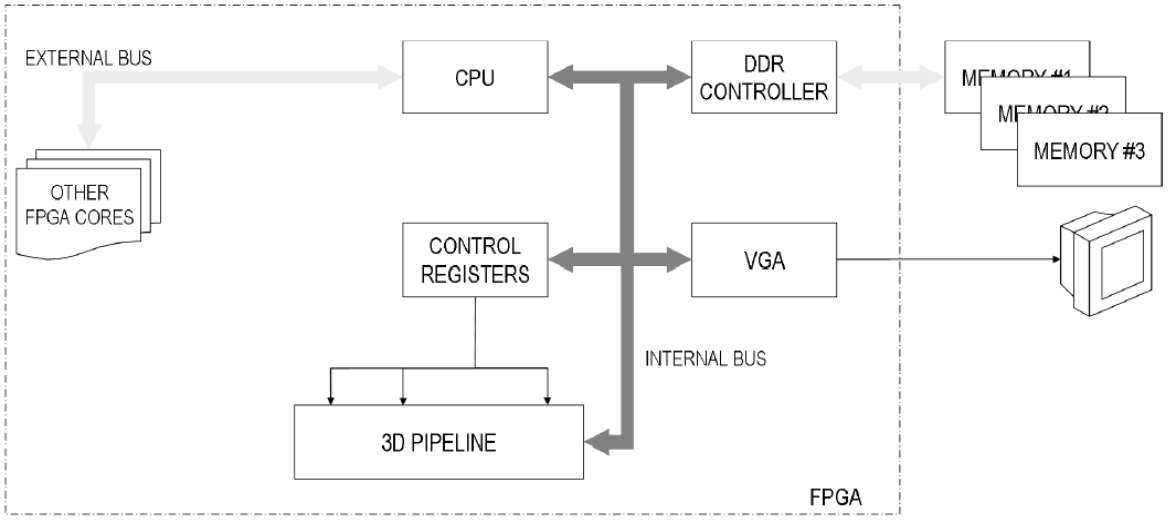


Figure 13: System overview of 3D graphics system [59].

The CPU handles all communication to the external and internal buses and also controls the system. The VGA controller reads the data out of the external DDR memory and formats it for display. The more interesting part of this research lies in the 3D pipeline, which is responsible for building raster data in memory. The 3D pipeline, which is shown in Figure 14 as a block diagram, roughly implements all of the pieces of the OpenGL pipeline. However, it is a bit unclear how the texturizer and pixel shader could work before the primitive drawer has actually generated the fragments or pixels. Because the 3D pipeline module also has to share the external memory with other system resources, the output of the 3D pipeline circles back to the internal bus shown in the system overview diagram.

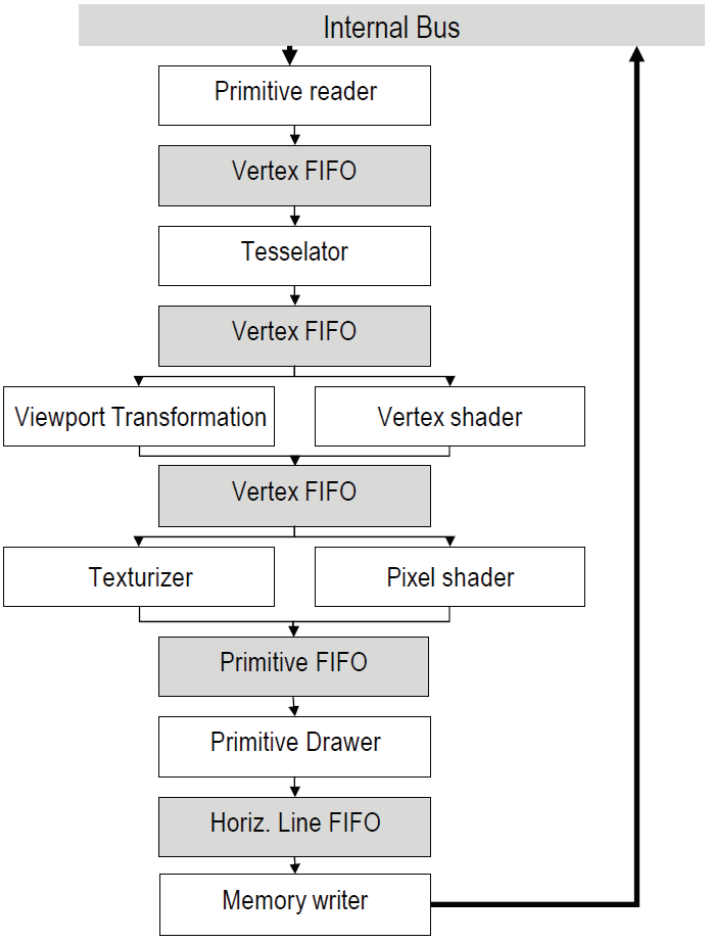


Figure 14: Block diagram of 3D pipeline module [59].

For the actual implementation of this design, the researcher attempted to use open-source IP cores as much as possible. The cores were stitched together with the Wishbone interface that is prevalent among cores found on the opencores.org website. The researcher spent so much time analyzing and integrating the open-source IP cores that the actual 3D pipeline was not even implemented. The researcher concluded the paper with results that show the CPU, DDR controller, and VGA cores all working appropriately. Thus, although the platform for building the 3D pipeline exists and is

ready for further research, the block that is missing is a key component that provides the ingenuity, and it is the focus of this current thesis.

To avoid the problems created by patching together public IP cores, it can often be more effective to design a new FPGA from the ground up. A 3D graphics accelerator was developed [60] in 2005 based upon the Nios II soft core processor available from Altera. This processor, used in conjunction with some limited hardware acceleration, provided the researchers with a stable platform for 3D graphics generation. The top-level block diagram is shown in Figure 15.

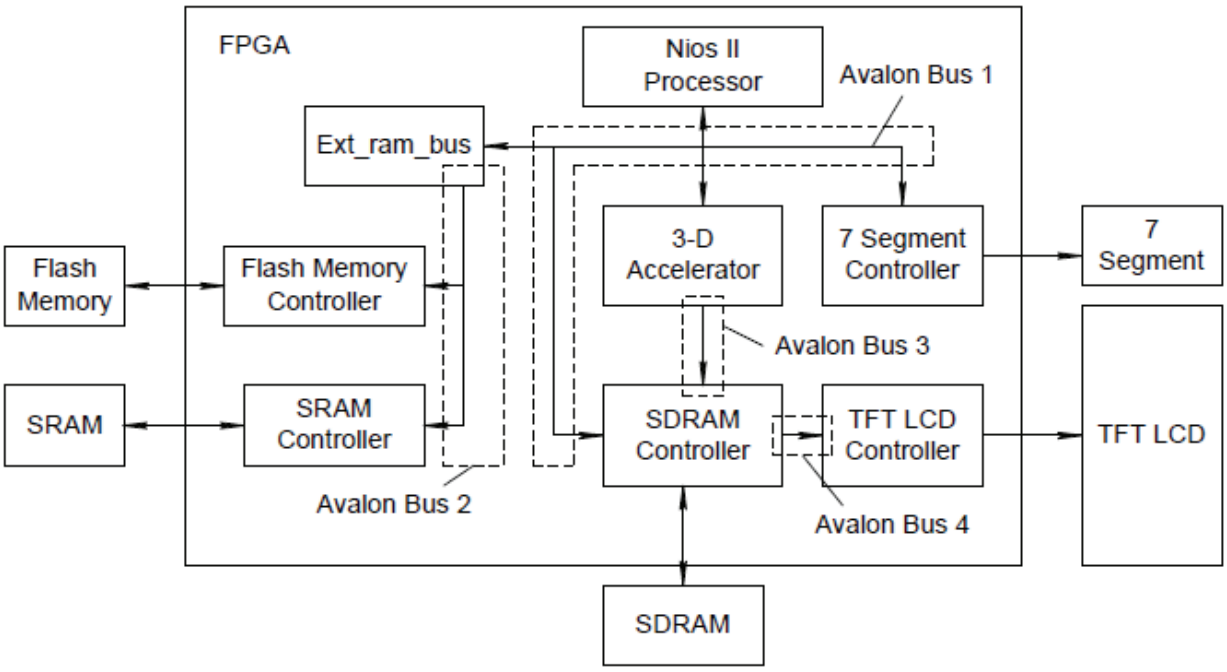


Figure 15: Top-level block diagram of 3D graphics accelerator [60].

The researchers segregated the OpenGL ES pipeline into two sections, with the vertex processing in software and the pixel processing in hardware. The overall processing performance of this architecture was not nearly fast enough for even a

consumer product. In an effort to improve performance, the authors plan to continue developing hardware modules to replace some of the software-based implementations. However, just moving the operations from software to hardware will likely yield only a linear improvement. By taking advantage of parallelism and hardware programmability instead, the researchers could most likely see an exponential improvement in performance.

In 2007, researchers focused back down on just the implementation of vertex shader functionality in the FPGA [61]. The FPGA vertex shader is viewed as a sort of hardware co-processor for use in embedded systems, and it includes coordinate transformation, illumination, clipping, projection, and floating-point conversion. For their tests, the researchers used a Virtex 4 device with an embedded PowerPC 405 processor. The block diagram is shown in Figure 16.

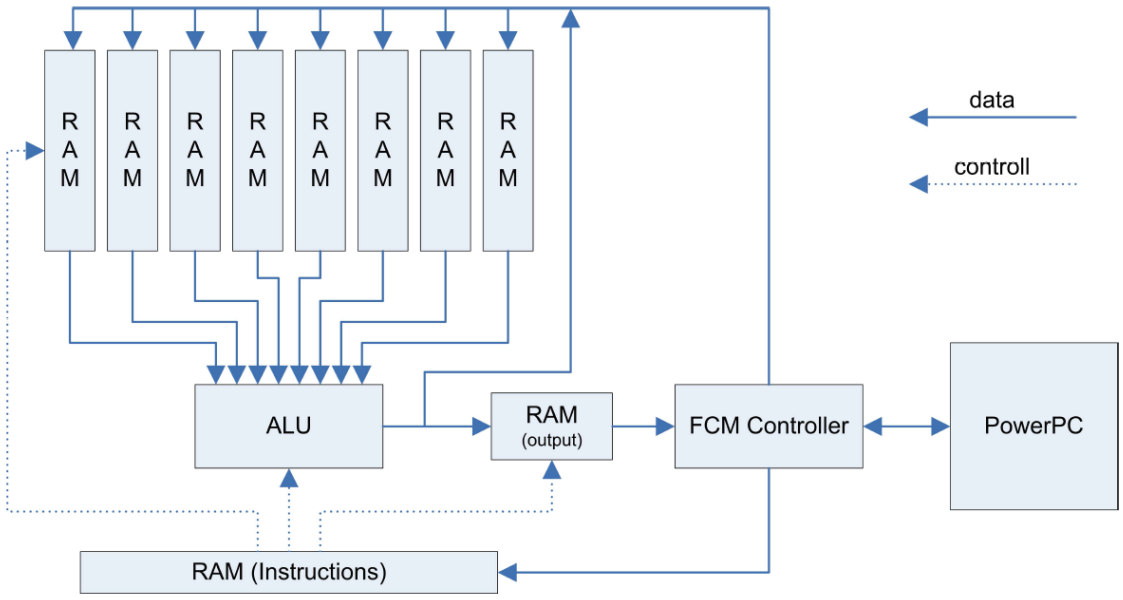


Figure 16: Block diagram of vertex shader co-processor [61].

The ALU supports nine different commands that are all useful during the vertex shading operations. Each vertex shader program is compiled with the DirectX SDK and then translated into the instruction formats recognized by the ALU. The results show that the hardware acceleration of the vertex shader was able to improve the processing speed by 3-6 times, depending on the number of vertices being processed. This research proves that GPU algorithms can be effectively mapped to FPGA resources with a substantial improvement over the software-only implementation. The performance comparison between an FPGA-based GPU implementation and an actual ASIC GPU still remains a question.

2.5.2 Commercial Offerings

In addition to academic research, several companies have also recognized the market need for FPGA-based graphics processing and have developed their own solutions. The market leaders, Altera and Xilinx, have both partnered with third-party IP core companies to provide graphics processing cores. In addition, Actel published a paper as early as 1994 that describes a very crude initial attempt at graphics processing in an FPGA that would now be considered primitive [62]. Actel basically developed a memory controller that pulls pre-defined video images out of an EPROM with a variable rate and direction. The design used an ACT 3 device with a maximum capacity of 10k gates, running at 50 MHz.

Altera and Xilinx, on the other hand, have let IP cores specialists design the graphics processing cores for them. Altera has partnered with TES Electronics Solutions to offer the D/AVE core in both 2D and 3D versions. The D/AVE 2D core focuses on

visually pleasing graphics for dashboard, navigation, and on-screen display (OSD) applications [63]. Figure 17 shows how D/AVE could be used in an FPGA GPU system. D/AVE provides multiple Avalon interfaces along the graphics pipeline. Avalon is Altera's switched fabric interconnect that allows seamless integration with Altera's SOPC Builder tool literally in a push-button design flow.

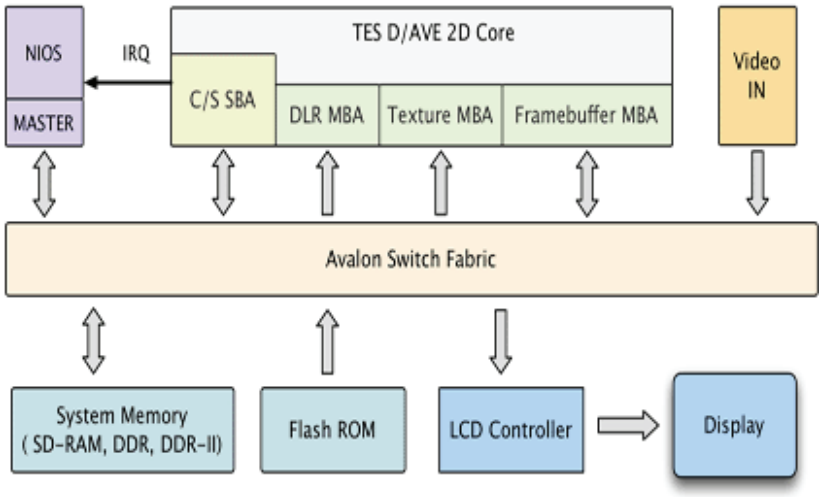


Figure 17: D/AVE 2D in an FPGA GPU system [63].

D/AVE 2D is available in two different variants – a "TS" suffix denotes the standard version and a "TL" suffix denotes the light version. The standard version can process one pixel per clock cycle, but the TL version requires four clock cycles per pixel, presumably with a corresponding reduction in the number of required logic elements. There is no OpenGL driver or interface provided, but D/AVE is well setup for using the Nios as a host CPU for the graphics system since the Avalon switch fabric is available.

For higher performance applications, TES Electronic Solutions has also developed D/AVE 3D for either FPGA or ASIC instantiations [64]. This core was

developed mainly for embedded, automotive, and infotainment applications. The block diagram in Figure 18 shows that an AMBA or Avalon interface is provided to communicate with the host CPU. The core is capable of generating lines, triangles, and quadrangles as native primitives. The major selling point of this core is that it does support OpenGL ES 1.1, which is the standard for graphics functionality in an embedded application. This standard host interface allows an FPGA with D/AVE 3D to function as a standalone GPU if desired.

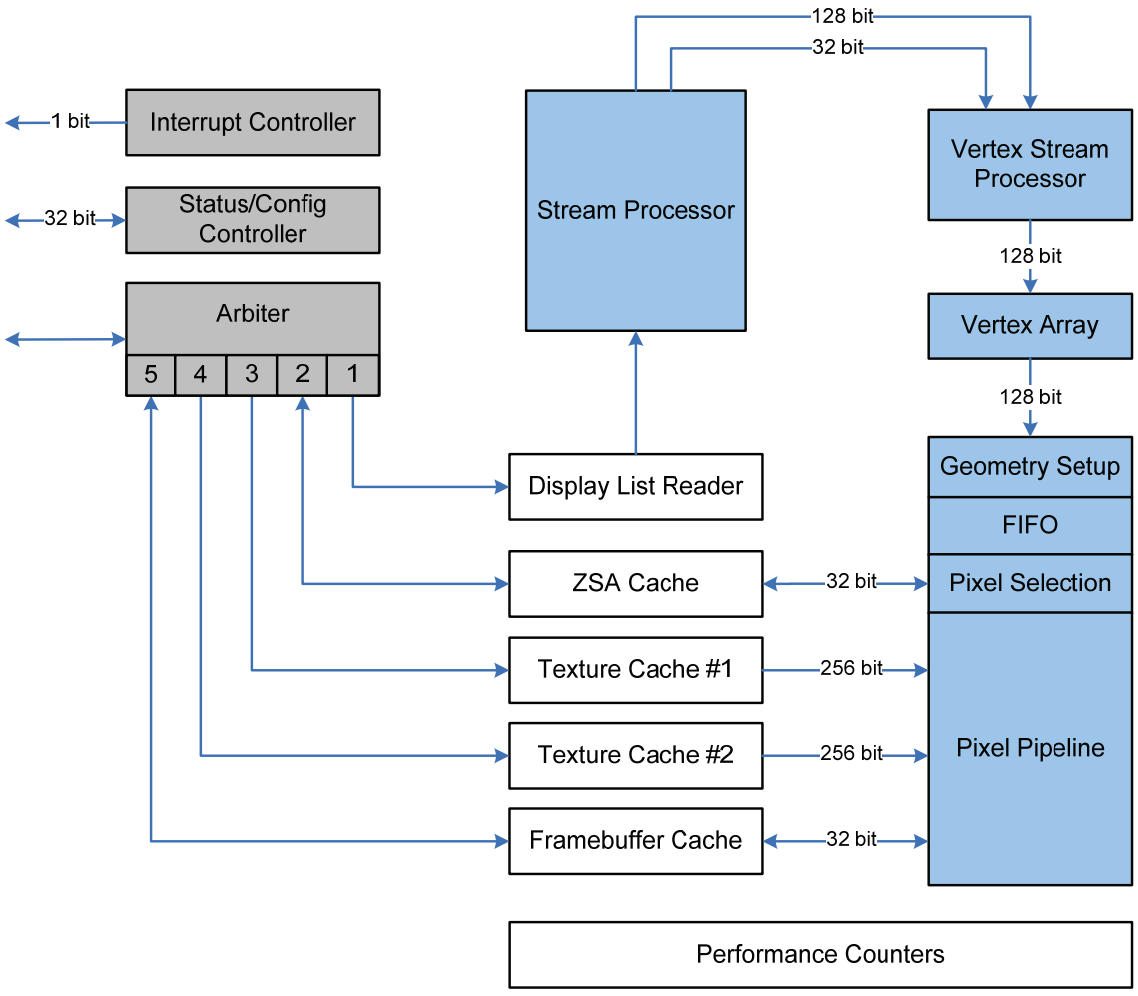


Figure 18: Block diagram of D/AVE 3D core (based on [64]).

Xilinx has partnered with Xylon (developer of logicBRICKS^{TM}) to offer the logi3D core, which is a scalable 3D graphics accelerator. Xilinx has exclusively marketed this core for automotive applications, but it could be used elsewhere as well because it is also based upon the OpenGL ES 1.1 API. The block diagram, shown in Figure 19, shows the unique approach that actually includes a MicroBlaze soft-core processor as part of the core [65].

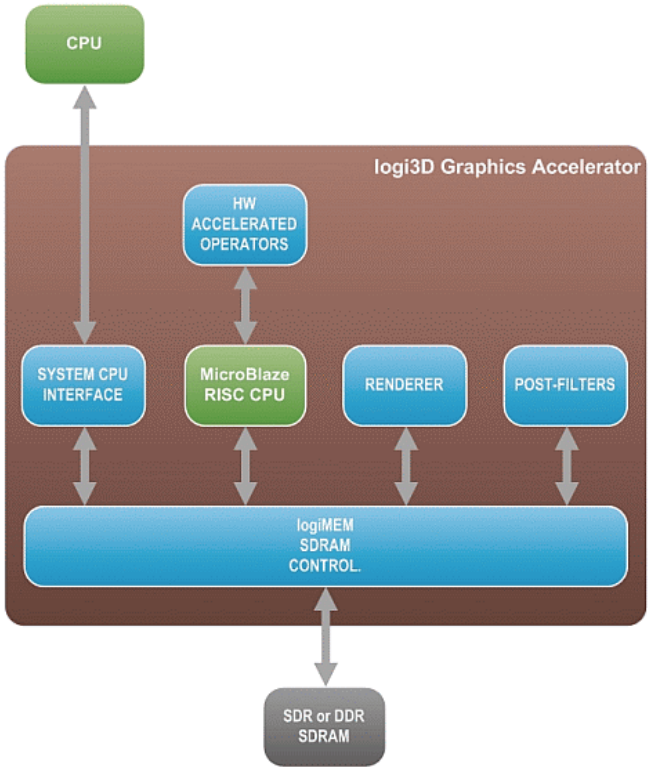


Figure 19: Block diagram of logi3D graphics accelerator [65].

The System CPU interface is based upon OpenGL ES 1.1 and can communicate with an off-chip CPU or another internal MicroBlaze CPU. The geometry stage of the graphics pipeline is shared between software that executes on the internal MicroBlaze

and hardware accelerations that offload matrix multiplication, frustum calculation, etc. The final stages of rendering and post-filtering (fog) are implemented completely in hardware. Each stage of the graphics pipeline is interconnected with the logiMEM IP core which provides an interface to the external SDRAM or DDR SDRAM. Although logiMEM does allow multiple system ports, it does not offer any control over arbitration, priority scheduling, or dedicated bandwidth options. Thus, although the graphics pipeline may be adequate for some applications, the memory bandwidth may become an issue especially for higher resolutions because the interface has not been optimized for a GPU.

CHAPTER 3

FPGA GPU ARCHITECTURE

After researching FPGA capabilities and existing GPU architectures, a new GPU architecture has been developed that is specifically targeted towards FPGAs [66]. This architecture provides basic graphics functionality that exploits the flexible fabric and unique structures of an FPGA, and it also avoids FPGA limitations in order to produce performance suitable for many applications. The architecture is modular and scalable to allow its use in a number of different applications, and Table 1 describes its high-level features.

Table 1. Architectural features.

Feature	Description
Host interface	PCI (32-bit, 66 MHz) or serial (RS-232)
Rasterization	Points, lines, and triangles
Shading	Flat or smooth
Color blending	Fragment colors are set by interpolation between vertices only (no textures)
Scissoring	User-defined and screen region
Alpha blending	A variety of factors based on the following equation: $C_O = SC_S + DC_D$
Output	32-bit color Adjustable resolution (VGA, SVGA, or XGA) Adjustable refresh rate (60 Hz, 30 Hz, or 20 Hz)

This architecture provides basic graphics processing capabilities suitable for applications including industrial displays, automotive displays, avionic displays, and other embedded products. In many industrial or embedded applications, the host

application generates menus or graphical overlays by specifying vertices in screen coordinates with defined colors. Typically such applications do not require full 3D textured rendering. As such, this basic architecture does not include vertex processing because coordinate transformations and lighting calculations are not needed. Based upon commands and vertex data (color and position) from the host application, the FPGA can generate anti-aliased points, lines, or triangles. Smooth shading and alpha blending are also incorporated to produce more realistic imagery. The architectural approach for the FPGA-based GPU is the subject of a pending patent [67].

3.1 Overview

The overall structure of the GPU architecture is motivated by the OpenGL pipeline, with distinct blocks assigned along the pipeline for specific functions. The architecture, shown in Figure 20 as a block diagram, establishes a flexible-function pipeline as opposed to the typical terms of fixed-function or programmable. The flexible-function pipeline, because of its modularity and re-programmability, can be easily modified or scaled up or down to meet specific performance requirements from many different applications. The re-programmability provides a key advantage over traditional fixed-function pipelines because it allows customizations and adjustments to permit functionality that was previously not possible. As an example, a unique alpha blending algorithm could be implemented very quickly and incorporated into the FPGA by modifying only one module. This capability is simply not possible with fixed-function GPU pipeline architectures, although it is available in the most recent ASIC GPU architectures with programmable pipelines. However, the ASIC GPU's extreme

programmability also comes at the expense of power, cost, and long-term device availability.

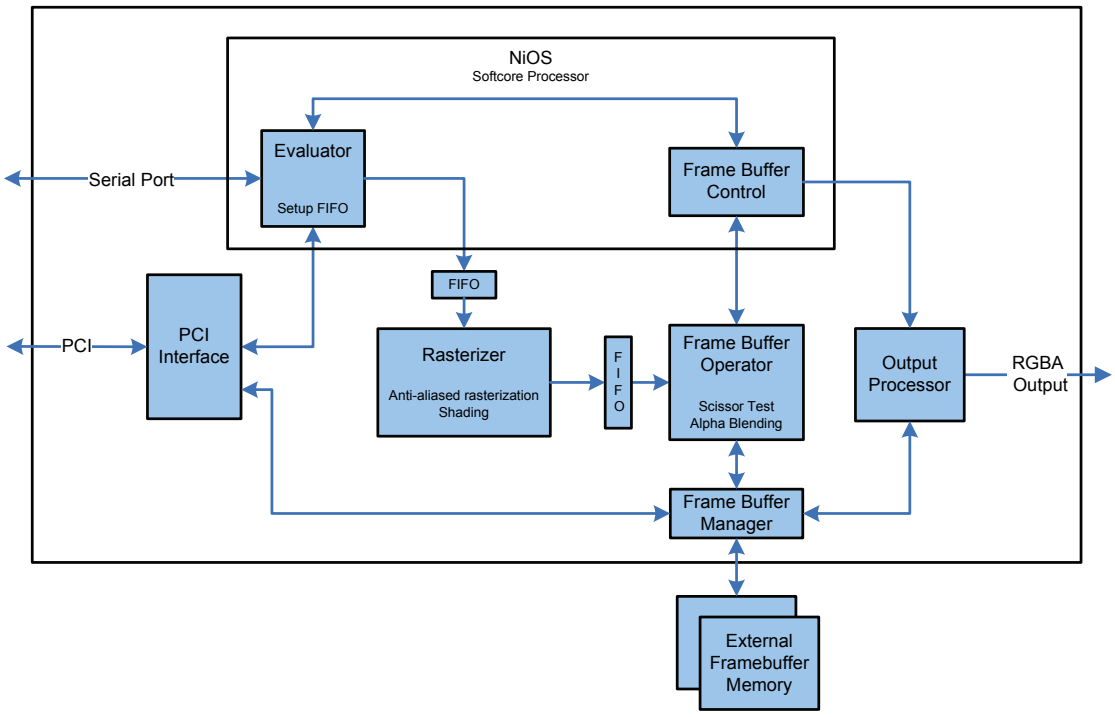


Figure 20: Architectural block diagram.

Several other architectural concepts were also explored that relied more upon a soft-core processor to handle much of the rendering. However, previous research [60] that depended upon soft-core processors resulted in lackluster performance with limited scalability. To accommodate requirements from many different markets, the soft-core processor in this architecture does not perform any of the actual graphics processing. The functionality in the GPU pipeline is all accomplished in true hardware modules that were custom designed for graphics processing. An overall block diagram from the Altera Quartus2 RTL Viewer tool is shown in Appendix B.

Much of the performance focus in this architecture is on the acceleration of the Rasterizer, which uses a novel packet processing platform discussed later. The Nios II CPU, an embedded soft-core processor from Altera, receives graphical commands and data from the host. The host interface software in the Nios II sets up the first pipeline FIFO with the necessary data, and it also controls high-level parameters such as resolution and frame rate. To maximize graphics throughput, the Nios II CPU only sets up the pipeline and controls some of its timing, and it is not involved in later stages of the GPU pipeline.

The hardware pipeline is based upon several scalable modules with FIFOs nested between them to provide buffering and diagnostic capabilities. The fill level of each FIFO is used to characterize pipeline inefficiencies for further optimizations. The FIFOs use the FPGA's on-chip memory blocks for minimum latency. After the initial pipeline optimization and integration phase, the FIFOs are used to provide flow buffering to compensate for the asynchronous or unpredictable input from the host. The FIFOs contain vertex or fragment data as well as all commands that affect the processing in the pipeline. The commands remain sequenced together with the graphics data to ensure that the commands are executed at the correct time relative to the vertices or fragments being processed.

3.2 Basic Graphics Functionality

This architecture was intentionally developed with five distinct modules so that the performance or functionality of any one block could be easily adapted for different applications. The Evaluator is a software module that can be easily customized for

different host interfaces or drivers. The Rasterizer employs a packet-processing platform to generate fragments in a very unique way. The third module, the Frame Buffer Operator, uses parallelism, DSP blocks, and on-chip memory to test and blend four fragments at once. The Memory Arbiter uses a state machine to handle priorities with the DDR2 memory core. And finally, the Output Processor uses a large FIFO to ensure that the data is transmitted with correct timing despite the different clock domains. Each of these five major modules is explained in more detail in the following sections.

3.2.1 Evaluator

The Evaluator serves as the host interface to receive graphical commands or data and format them for use by the GPU pipeline. A Nios II soft-core processor from Altera was chosen to implement the Evaluator to take advantage of existing intellectual property for the UART serial interface. While the serial interface is not used as the primary means of transferring data from the host application, it is extremely useful for pipeline control as well as analysis and diagnostics. The throughput and/or status of any module in the pipeline can be quickly queried and determined. The Nios II soft-core processor uses general-purpose I/O (GPIO) signals to retrieve statuses from the individual blocks in the pipeline. In addition, a bitmap capture of the entire frame buffer contents can be retrieved with the use of the Evaluator. When commanded, the Evaluator will transmit out one pixel at a time over the serial interface, back to a host computer. A separate utility called TXT2BMP was created to run on the host computer and convert a text file of pixel data into the bitmap format. This same utility was also used during FPGA

simulation to visually validate the frame buffer contents, and it is discussed in more detail in Chapter 4.

A graphical user interface (GUI), shown in Figure 21, was developed to aid in communication with the GPU pipeline through the Evaluator's serial interface. The software for this GUI was designed concurrently with the Evaluator's software so that the command set and data format on the serial interface would match.

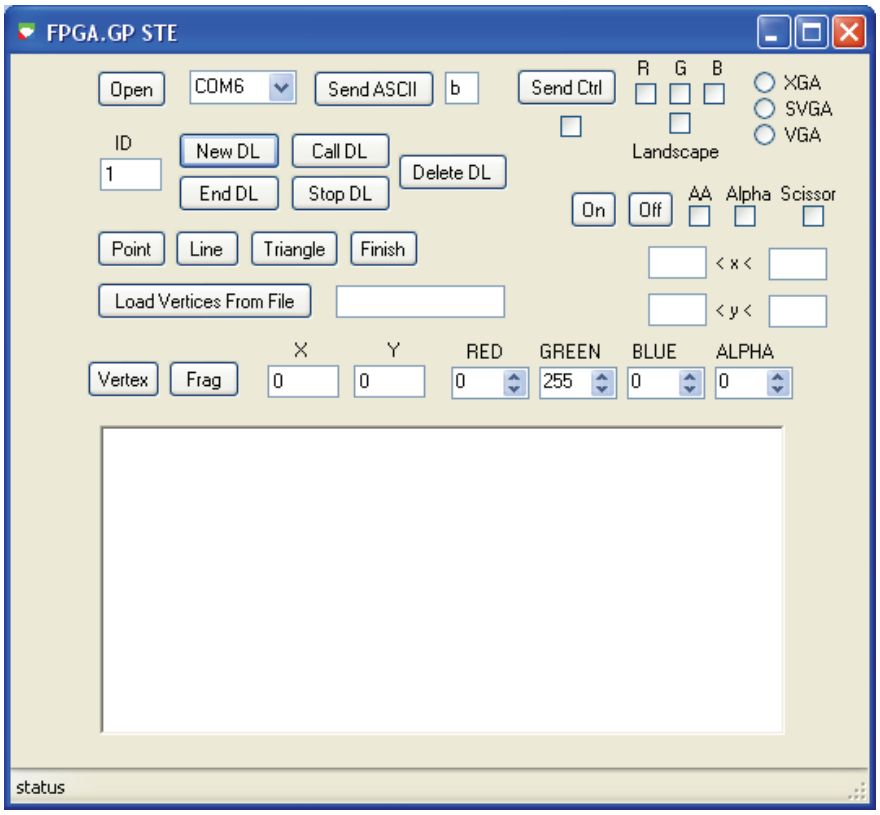


Figure 21: Graphical user interface for testing basic graphics processing functionality.

The GUI can control resolution, background color, orientation (landscape/portrait), anti-aliasing, alpha blending, and scissoring. In addition, the user can specify primitives (point, line, or triangle), vertices (with 32-bit RGBA color), and define display lists. Display lists are pre-defined graphical commands and data that can be called at a later time. In order to aid in the validation of this GPU architecture, the Evaluator stores the display list information in the FPGA's internal block memory. When a display list is called by the GUI, the Evaluator plays back the predefined sequence of commands and data every frame. Thus, a continual playback of display lists simplifies the performance analysis because the primitives are rendered every frame at a 60 Hz rate without the GUI having to send commands continually.

The primary responsibility of the Evaluator is to set up the graphics commands and vertex data into structures that are written into the FIFO buffer. All commands, vertices, and fragments flow through the pipeline in sequence so that the commands that change the pipeline are processed at the correct time relative to the vertices and fragments. There are multiple FIFOs along the pipeline to allow for storage while the downstream modules are continuing to process previous data. The sizes of the FIFOs were adjusted during integration based on the performance characteristics of each processing module. The data in each FIFO is composed of ten bytes, with the first byte defining the type of data. For different applications that use a unique interface or driver on the host CPU, only the Evaluator module would have to be modified. The pipeline data format is flexible enough that it can remain the same so that the downstream pipeline modules would not need to be altered. The source code for the Evaluator module is included in Appendix C.

Because the GUI and Evaluator's serial interface allowed for the GPU pipeline to be exercised, analyzed, and evaluated, the PCI interface was not actually instantiated in the initial architectural development that took place on an Altera Stratix III development kit. However, such a high-bandwidth interconnect was implemented in the avionics application that is described in Chapter 5. For this basic graphics processing, the PCI interface would simply map to internal registers in the Evaluator, and then the Evaluator would format the data for insertion into the GPU pipeline. The instantiation and integration of this interface was not critical to the initial evaluation of the GPU's architectural details.

3.2.2 Rasterizer

The Rasterizer is the most significant module in the GPU pipeline, and it also provides the interesting research contributions as its architecture is the topic of a pending patent [68]. The Rasterizer is responsible for interpolating between vertices to generate lines and filling between triangle edges to generate triangles. The output of the Rasterizer is a stream of fragments that may eventually become pixels in the frame buffer. Each fragment has a 32-bit color value as well as a 20-bit XY coordinate. The Rasterizer can produce shaded and/or anti-aliased primitives.

Several architectural options were considered for the Rasterizer, including just building a purpose-built hardware block to generate fragments based upon input vertices. However, to provide easy scalability for many different performance levels, the rasterization engine exploits a high-performance packet processing engine from Altera. This packet processing framework uses a task/event relationship between a multithreaded

soft processor and hardware acceleration blocks. Although this engine has previously only been used in packet processing applications, this FPGA-based GPU pipeline provides a unique set of requirements that make the packet processing perspective an appropriate one. This section first provides an overview of the packet processing engine, then explains how the engine is used for rasterization, and finally presents some data concerning scalability and extendibility. More detailed RTL and state machine diagrams of the Rasterizer are included in Appendix D.

3.2.2.1 Packet Processing Platform

The packet processing platform has a very small footprint but contains all of the necessary controls and interfaces to attain high performance. The soft processor uses hardware multithreading to execute up to eight threads at once. There are eight sets of registers and program counters so that each thread operates independently of the other threads and the registers can be switched in and out with zero latency. The performance of the packet processing is maximized with the use of hardware accelerators called event modules.

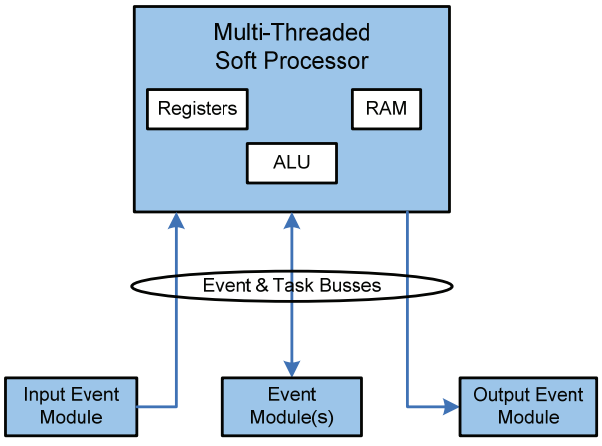


Figure 22: Packet processing platform block diagram.

A high-level block diagram of the packet processing platform is shown in Figure 22. The hardware input event module first calls a task in the soft processor with a new packet ID. The processor can then offload processing to hardware by passing the packet ID to an event module. The packet alternates back and forth between hardware and software until it has completed processing, at which time an output event module releases the packet ID. Up to 64 packets can be in the system at one time. The software can be written in either C or assembly, but the algorithms written in assembly provide for maximum performance.

3.2.2.2 Rasterization Engine

The migration of the packet processing platform from its intended networking application into a graphics rendering application provides novelty to this research. The goal of the rasterization engine is to generate fragments (in-process pixels) as quickly as possible after being given the two endpoints of a line or the three vertices of a triangle. Anti-aliasing can be enabled or disabled, which ultimately results in two different algorithms merged into one process. The rendering of a triangle is essentially an extension of line rendering, which is best accomplished with the Bresenham and Wu algorithms [69, 70].

The Bresenham and Wu algorithms were first implemented in a MATLAB model to allow analysis and also to provide a comparison once the VHDL algorithm was implemented. An analysis of the line rasterization computations allowed a separation between setup and recursive portions. In this architecture, the multithreaded soft processor is responsible for much of the setup portion of the algorithm, and hardware

acceleration blocks implement the recursive portion. The division computations required in the line setup are offloaded to hardware as well. Figure 23 shows how the rasterization functionality is partitioned within the packet processing platform.

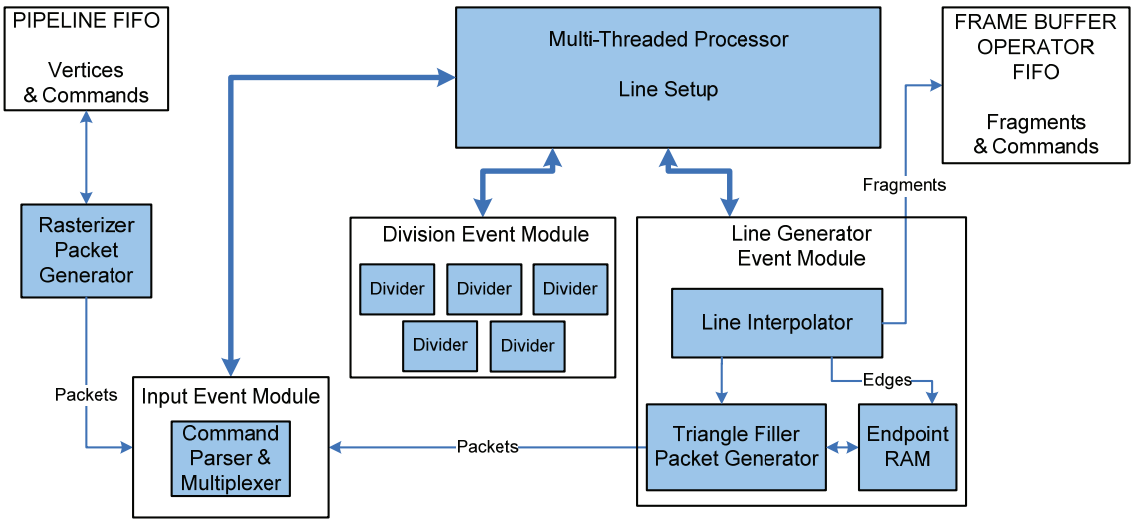


Figure 23: Rasterizer block diagram.

The Rasterizer Packet Generator receives commands and vertices from the Pipeline FIFO and creates Avalon-ST packets that the input event module can accept. The Avalon-ST interface has been standardized by Altera to provide an extremely flexible, high-bandwidth streaming protocol. The input event module parses the packet and issues the initial setup task in the multi-threaded processor. After the processor determines which axis is dominant (has a larger change in value) in the line, it sequences the vertices so that the vertex coordinates increase along the dominant axis. Next, the processor offloads the slope and color gradient calculations to a hardware event module. The slope and gradients are sent back to the processor via another task call. The processor then collects all of the necessary data and issues an event call to the hardware

line generator event module, where the actual fragments are generated with the Bresenham or Wu algorithms. At the completion of the line, the event module issues a task back to the processor to provide notification that the packet has completed processing.

There are a couple different anti-aliasing techniques as well as scan conversion methods, which are the processes for converting each polygon into a series of horizontal lines. Some anti-aliasing techniques such as full-scene anti-aliasing or super-sampling are somewhat independent of the rasterization, but primitive-based anti-aliasing actually happens during the rasterization. For anti-aliased triangles, only the edges need the special treatment that results in a smoothed appearance. Full-scene anti-aliasing is wasteful for large triangles because the ratio of interior fragments to edge fragments is so low. Instead, a more optimum method is to anti-alias only the outer edges of the triangle. There are two ways to generate anti-aliased triangles in this manner.

The first anti-aliased triangle algorithm, shown in Figure 24a, fills horizontal lines along sub-scanlines and produces a triangle that is actually at a higher resolution. The sub-scanlines are then merged back down to the desired resolution by generating a coverage value for each x-coordinate in the line. This coverage value determines the weighting or shading of the edge fragments on each line. The second anti-aliased triangle algorithm, shown in Figure 24b, draws the anti-aliased edges first and then goes back to fill in the triangle interior with horizontal lines. This algorithm is more efficient for large triangles and actually takes advantage of the FPGA's internal block RAM as described in the following paragraph. Both anti-aliased triangle algorithms were implemented and evaluated, and the second algorithm was selected for this basic graphics architecture.

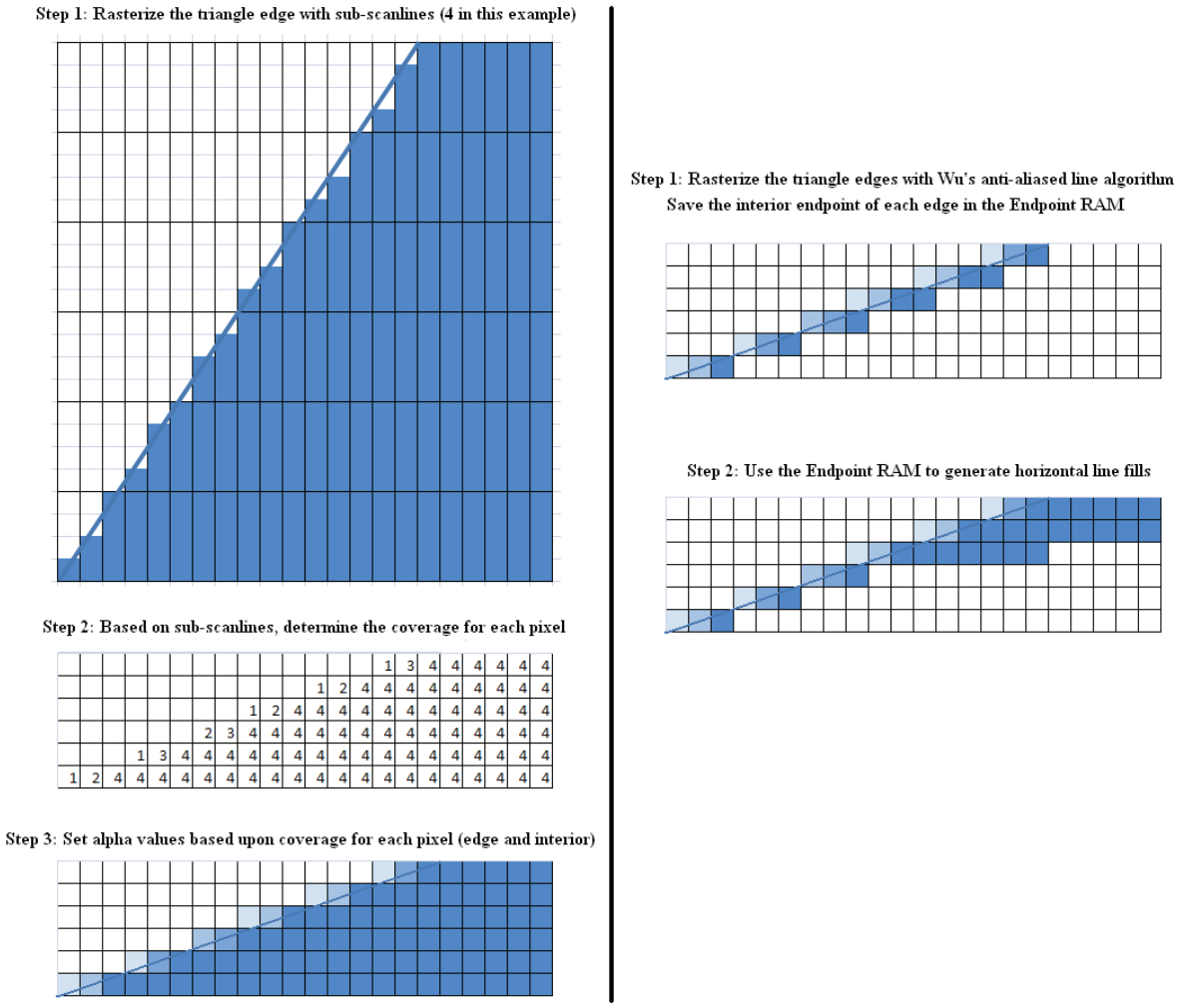


Figure 24: Anti-aliased triangle rasterization algorithms.

To rasterize the triangle, the soft processor first determines which side of each line is on the outer edge of the triangle. This characteristic is important when anti-aliasing triangles so that only the outer edges are smoothed. Otherwise, artifacts would be generated during the triangle filling. The steps for generating a triangle are shown in Figure 25, where each line is numbered to indicate the order of operations and all of the shaded boxes indicate hardware functions. First the triangle command packet is parsed

and sent to the processor. The packet then bounces between hardware and software to create the three triangle edges. As the edges are generated, the endpoints for each interior horizontal line are saved into the Endpoint RAM, whose format is shown in Figure 26. At the conclusion of the third edge, the embedded Endpoint RAM is searched to find which horizontal lines need to be created to fill the triangle. Finally, new horizontal line command packets are generated and sent back through the Rasterizer.

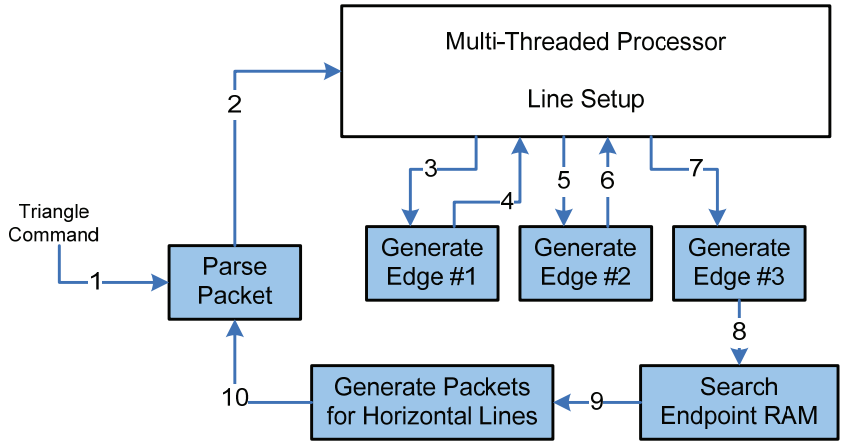


Figure 25: Sequence of steps for triangle generation.

Each y-coordinate in the screen resolution has ten entries in the endpoint RAM, and each entry is initialized with all zeros. The y-coordinate provides an index for all of the possible horizontal lines in the screen resolution. The RAM holds the x coordinates and RGBA values for the two endpoints of each horizontal line. In the example shown below, the pre-processing block generates a new horizontal line from (450, 232) to (503, 232) that will blend from a mostly red color to a mostly green color.

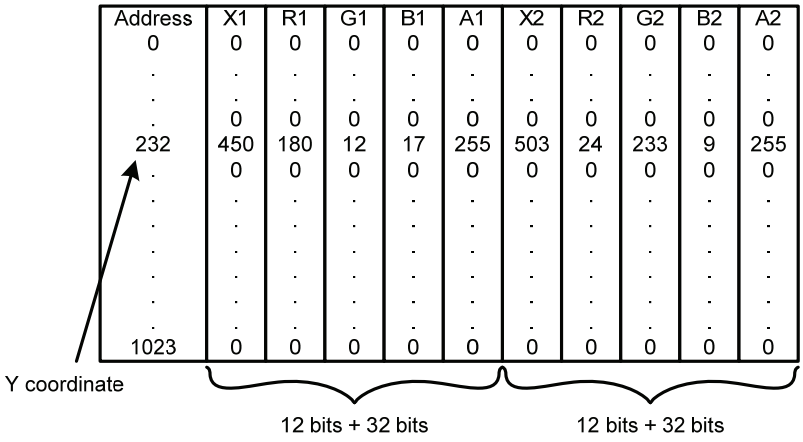


Figure 26: Endpoint RAM format.

3.2.2.3 Rasterizer Scalability

The Rasterizer has a rather simple goal of creating fragments based upon input vertices. However, this architecture's Rasterizer is unique in that it is highly scalable to produce significantly more fragments with very little additional effort required. The performance of the Rasterizer is directly dependent on the operation of the Draw Event Module, whose performance bounds the overall throughput of the Rasterizer. Thus, the Rasterizer only uses one thread out of the eight threads possible because the processor is always waiting on the Draw Event Module to accept another event. To double the fill rate of the Rasterizer, the Draw Event Module can be replicated as shown in Figure 27. This small change allows twice as many fragments to be generated. Figure 28 demonstrates that up to eight Draw Event Modules could be added to the Rasterizer, which would maximize the performance. The actual performance differs from the ideal performance because of bus arbitration issues with the event and task buses internal to the packet processing engine. In addition to the extra Draw Event Modules, a post-processor

block also has to be added to the Rasterizer to collect the fragments from each line generation and write them into the Frame Buffer Operator FIFO.

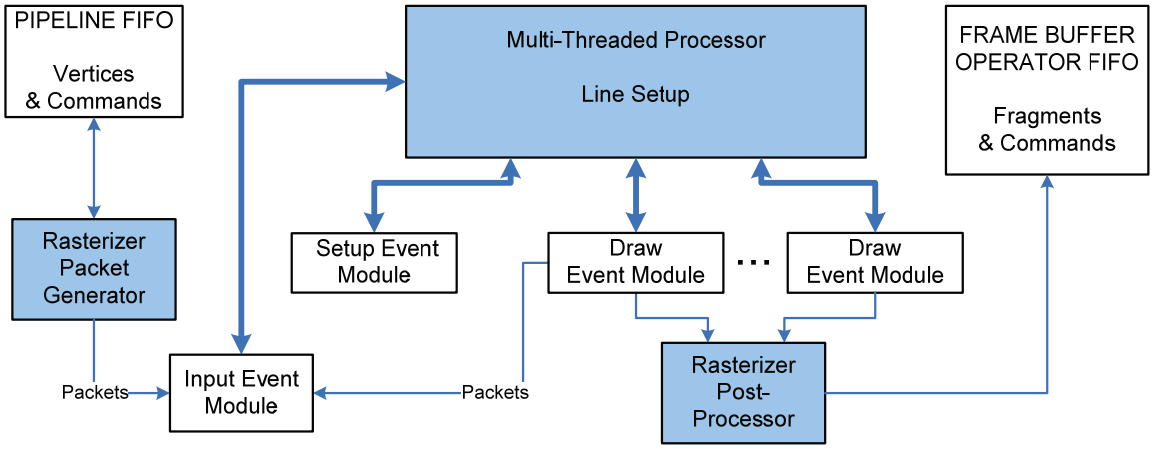


Figure 27: Rasterizer block diagram with multiple Draw Event Modules.

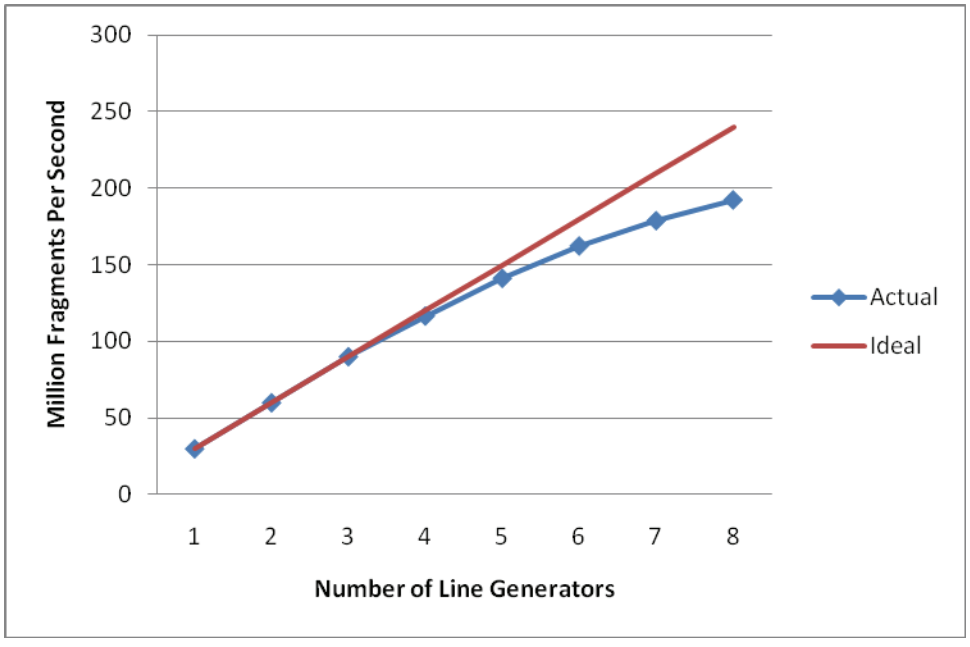


Figure 28: Rasterization scalability.

As the pipeline is made more parallel to accelerate performance, other synchronization issues develop. The OpenGL pipeline dictates that commands (state changes) and vertices/fragments must stay in order throughout the pipeline. When the processing is split out into parallel data paths, a mechanism must be developed to synchronize all of the configuration states. This issue was analyzed three different ways before a final approach was implemented. In the configuration tag approach, a small tag (8 bits should suffice) is added to each fragment that exits the Rasterizer. The tag is an index into a look-up table that is updated by the Rasterizer each time any of the state variables are modified. The Frame Buffer Operator then looks up the state configuration that is associated with the fragment and acts upon it accordingly. However, this configuration tag approach, while using the FPGA's fast internal block memory for the look-up table, requires extra clock cycles for every single fragment.

An alternate approach is simply to stall the Rasterizer whenever a state changing command enters the Rasterizer, and then to pass or act upon the command only after the Rasterizer is empty. This approach definitely decreases the performance, but only when commands are used often. If the pipeline is relatively stable (e.g., anti-aliasing is always on, or alpha blending is always off), then this approach might provide acceptable performance. However, a third approach compromises between tagging each fragment and stalling the pipeline. If the Rasterizer maintains all of the state variables and sends out state refreshing packets between each primitive, then each fragment is not delayed during processing and the Rasterizer can continuously process vertices. This approach works because commands will never enter the pipeline in the middle of a primitive. The

entire primitive will always have the same state, so the state only needs to be updated (at most) between each primitive.

The selection of a synchronization method is highly dependent on the expected behavior of the application. For this research, the stall approach was selected because state changing commands were not expected to enter the pipeline on a frequent enough basis to warrant the additional overhead of the alternate approaches. For applications that expect many different state changing commands, the configuration tag approach provides the most performance while still maintaining the strict ordering required by OpenGL.

3.2.2.4 Rasterizer Extensions

To optimize performance even further than the approximately 8x improvement that can be made with the packet processing engine, the Rasterizer could be easily extended to generate filled circles in addition to triangles. In OpenGL, a filled circle is typically rasterized with the triangle fan primitive, which is a sequence of triangles with adjoining sides. Each triangle in the fan can be thought of as a pie slice, as shown in Figure 29a. Each triangle is filled separately, and the end result is a filled circle. However, this approach is very time consuming because each radial line has to be rasterized and then each very small triangle is filled. In a typical horizontal line fill, the memory bandwidth is maximized because the memory addresses along the line are contiguous and burst memory accesses can be used. By using multiple filled triangles instead of a continuous horizontal line, the operation is broken into multiple steps and the bandwidth efficiencies of line filling are lost.

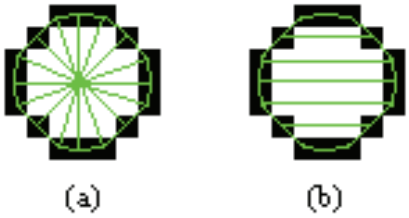


Figure 29: Typical and optimized circle rasterization approaches.

A much more efficient approach takes advantage of the Endpoint RAM structure that is built into this architecture to fill in polygons. In the approach shown in Figure 29b, the circumference of the circle is drawn with either a line strip primitive or even the Bresenham circle algorithm [71]. Using a line strip primitive means that the application provides each of the vertices along the periphery of the circle, and the Rasterizer draws very small line segments between vertices. Alternatively, Bresenham's circle algorithm only requires the application to provide the center and radius of the circle. This algorithm works with simple addition and rounding, very similarly to Bresenham's line algorithm. As the circumference is drawn (with either a line strip or Bresenham's circle algorithm), each of the fragments is loaded into the Endpoint Ram in a manner very similar to that of the triangle rasterization. When the circle circumference has been generated, the interior can be filled by stepping through the Endpoint RAM, drawing horizontal lines according to the endpoints specified.

The performance improvement, shown in Figure 30, is significant because as the circle radius increases, the number of memory accesses for each individual radial segment increases as well. Unless some memory addresses are adjacent, only four individual fragments can be written during each memory access. Thus, for a radius of 50 pixels, 13 memory accesses are required for the majority of the radial segments.

Alternatively, the optimized approach writes up to 32 pixels during each memory access by organizing the circle fill along horizontal lines.

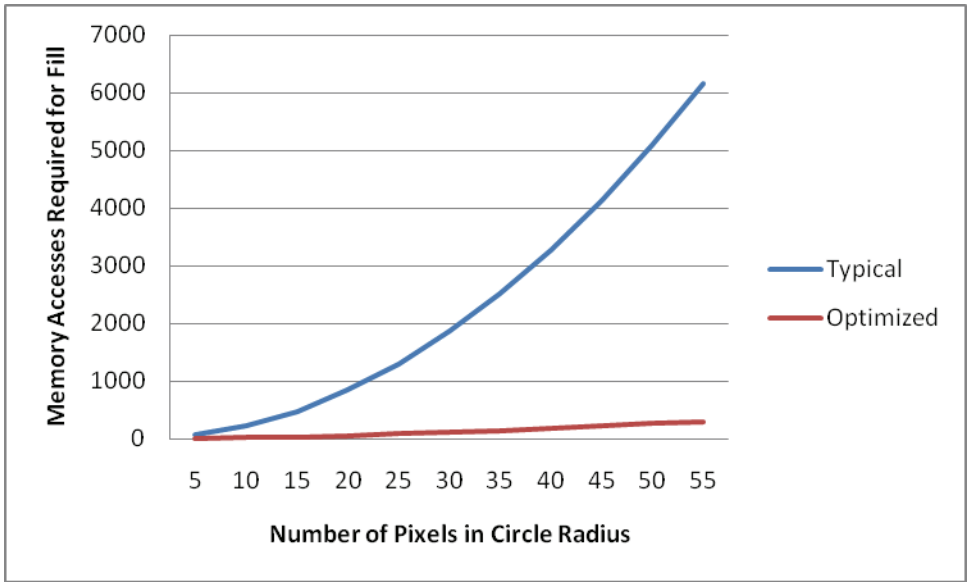


Figure 30: Comparison of typical and optimized approaches to circle rasterization.

3.2.3 Frame Buffer Operator

The Frame Buffer Operator (FBO) receives fragments from the Rasterizer and performs final tests and alpha blending on the fragments before they are stored in the frame buffer. There are actually two frame buffers – a front buffer and a back buffer. The displayed image comes from the front buffer while the pipeline generates graphics in the back buffer. Once the entire front buffer has been transmitted out, the buffers are swapped and the new back buffer is cleared before more fragments are stored. The detailed architecture of the FBO is shown in Figure 31. The FBO uses a state machine to sequence through the following tasks when a new fragment is received from the Rasterizer FIFO: scissor tests, pixel address translation, cache lookup, and possibly alpha

blending. More detailed RTL diagrams and state diagrams of the Frame Buffer Operator are included in Appendix E.

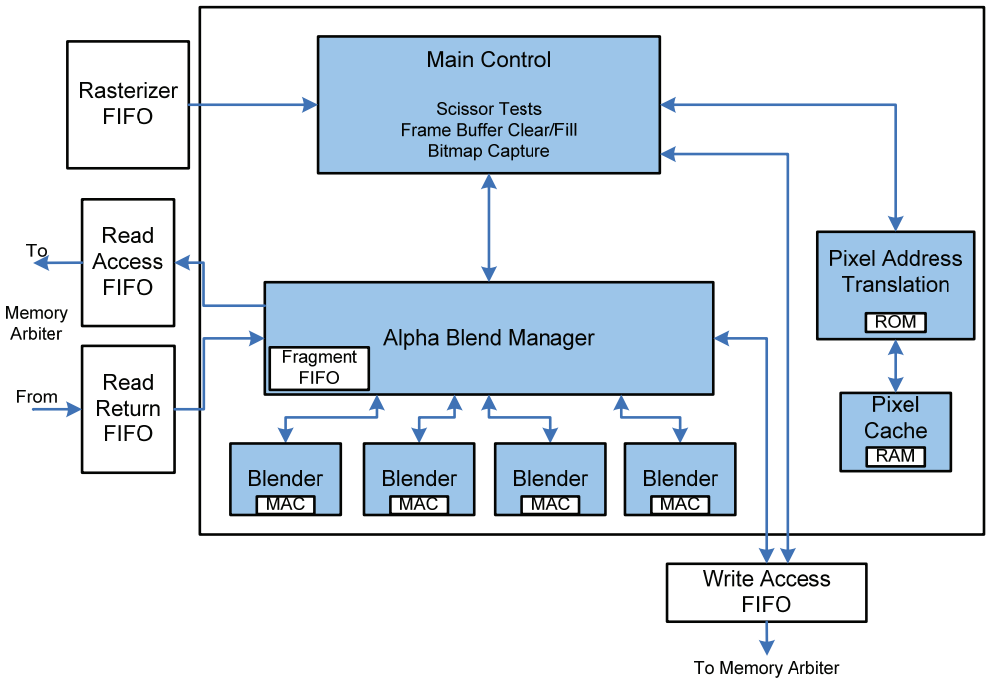


Figure 31: Frame Buffer Operator block diagram.

The scissor test first eliminates any fragments that fall outside the user-defined scissor regions. The FBO then uses a look-up table ROM (in the FPGA's on-chip memory) to accelerate the pixel address translation, which converts the fragment's x-y coordinates into a memory address. The basic equation for performing this translation is

$$\text{Address} = (X * \text{horizontal size}) + Y \tag{3.1}$$

The horizontal size depends upon the resolution, so this value is either 640, 800, or 1024, for VGA, SVGA, and XGA respectively. Thus, three separate look-up tables

are used to perform these multiplications. The contents of the tables are preloaded such that the address for the LUT is used as the X value, and the output of the LUT will be the address multiplied by the horizontal screen size. As an alternative, this LUT-based approach to the translation could be replaced with an approach that uses the FPGA's multiplier blocks. The choice of using LUTs or multipliers depends upon the resources that are available in the specific FPGA device.

Once the memory address is known, a pixel cache is used to determine if a fragment with the same address has already been generated in the current frame. This is a key contribution in the architectural research because it reduces the alpha blending latency from twenty clock cycles down to two clock cycles. The pixel cache uses on-chip memory that is 1-bit deep with 1024x768 locations (XGA resolution). A value of '1' in the cache location indicates that the corresponding pixel address in the frame buffer has previously been written in the current frame. If the particular pixel location has never been written, then it is unnecessary to read from that location during alpha blending. The pixel cache also has to be cleared every frame, whenever the frame buffer is cleared. The synchronized clearing ensures that the pixel validity in the cache is always correct.

This pixel cache contribution is very significant because memory bandwidth is a critical component that drives the performance of graphics processors. Table 2 shows that for various resolutions with graphics across 30% of the display, the memory access time for alpha blending can consume up to 98% of the frame period. When the pixel cache is introduced, the memory accesses are only required for a much smaller portion of the frame. A typical display could have about 20% of the pixels overlapping, and this would result in a maximum of 19.7% of the frame period being used for alpha blending.

Table 2. Alpha blending optimizations with pixel cache.

		No Pixel Cache				Optimized with Pixel Cache		
Resolution	**Total Pixels**	**Typical # Pixels in Graphics Overlay (30% of total)**	**Memory Access Time for Blending All Fragments (sec)**	**% of Frame Rate (60 Hz)**		**# of Actual Alpha Blended Pixels (20% of graphics)**	**Memory Access Time with Pixel Cache (sec)**	**% of Frame Rate (60 Hz)**
VGA	307200	92160	0.0038	**23.0%**		18432	0.0008	**4.6%**
SVGA	480000	144000	0.0060	**35.9%**		28800	0.0012	**7.2%**
XGA	786432	235930	0.0098	**58.9%**		47186	0.0020	**11.8%**
SXGA	1310720	393216	0.0164	**98.1%**		78643	0.0033	**19.7%**

Alpha blending is the process of blending a new fragment with one that is already in the frame buffer. There are several different factors that can be used in the basic alpha blending equation, which is $C_O = SC_S + DC_D$. In this equation, S and D are the factors, C_S is the new fragment color, and C_D is the old fragment color in the frame buffer. C_O is the color of the blended fragment that will be stored in the frame buffer. Regardless of the factors used, the FBO needs the color of the existing fragment in the frame buffer to perform the blending. The inherent nature of DDR2 memory introduces a significant latency to this read-modify-write transaction – unless the on-chip pixel cache indicates that an existing fragment is not in the frame buffer at the same address.

There are actually four alpha blenders because the memory controller has a maximum burst size of four reads, and a manager coordinates memory accesses to and from each alpha blender. When the manager issues a read request to the frame buffer, it stores that fragment's data and address in a local fragment FIFO. When the frame buffer data returns to the manager, the fragment's data is pulled from the local fragment FIFO and given to the next available alpha blender.

Each alpha blender performs the blending by using embedded DSP blocks for 8-bit x 9-bit multiplication. Four DSP blocks are required in each blender – one each for the red, green, blue, and alpha components. The use of sixteen DSP blocks for this functionality is an example of how this architecture takes advantage of the FPGA's structures to maximize performance while not having to use any of the FPGA's logic elements. Furthermore, the design complexity and latency are reduced from an implementation that uses either logic elements or memory to perform soft multiplication.

3.2.4 Memory Arbiter

As with other graphics processing architectures, the memory interface and arbitration in this architecture is a key contributor to high performance. A single DDR2 memory interface is used for both the front and back frame buffers, so the Output Processor, Frame Buffer Operator, and PCI Interface must all share this one resource. The memory interface uses Altera's High-Performance DDR2 Controller core which includes the memory's physical interface core. The core can operate up to 400 MHz but introduces a read latency of 21 clock cycles and a write latency of 13 clock cycles [72]. To compensate for some of this latency, the transactions can be grouped together in bursts of four. The burst capability allows four read transactions within 25 clocks instead of 84 clocks.

This memory architecture differs quite a bit from traditional GPU architectures because of the FPGA's relatively small on-chip memory and limited clock frequencies. Nvidia's Tesla uses an extremely wide 384-signal interface to 768-Mbytes of embedded GDDR3 RAM with a memory clock rate exceeding 1 GHz [40]. The FPGA's memory

architecture, on the other hand, must be crafted based upon the limitations imposed by the FPGA's capabilities. As a result, several arbitration methodologies were developed and analyzed before the final solution was implemented.

From a very high level, there are two ways to approach the issue of memory arbitration – using Altera's Avalon fabric or designing a custom arbiter. Altera has developed the Avalon system interconnect fabric (SIF) to handle all of the low-level details of on-chip interfaces. The SIF is automatically generated in Altera's SOPC Builder tool based upon how the Avalon master ports and slave ports are interconnected. Using this approach means that the design effort is much simpler, but it relinquishes all control and possibilities for performance optimizations. As an alternative, the memory arbitration could be custom designed in order to maintain control over the performance. When taking this more manual approach, two different arbitration schemes were investigated: credit-based or priority-based. A credit based scheme would assign a certain number of tokens to each interface so that the bandwidth was proportionately divided based upon relative importance. However, for this architecture, a priority-based algorithm was used to maximize performance by ensuring that the graphics pipeline was not stalled as a result of memory bandwidth conflicts.

In order to hide the latencies caused by the external memory interface, the proposed architecture uses several relatively shallow FIFOs. Figure 32 depicts the three FIFOs used between the FBO and the Memory Arbiter (MA) to hide this memory latency. More detailed diagrams of just the Memory Arbiter are included in Appendix F.

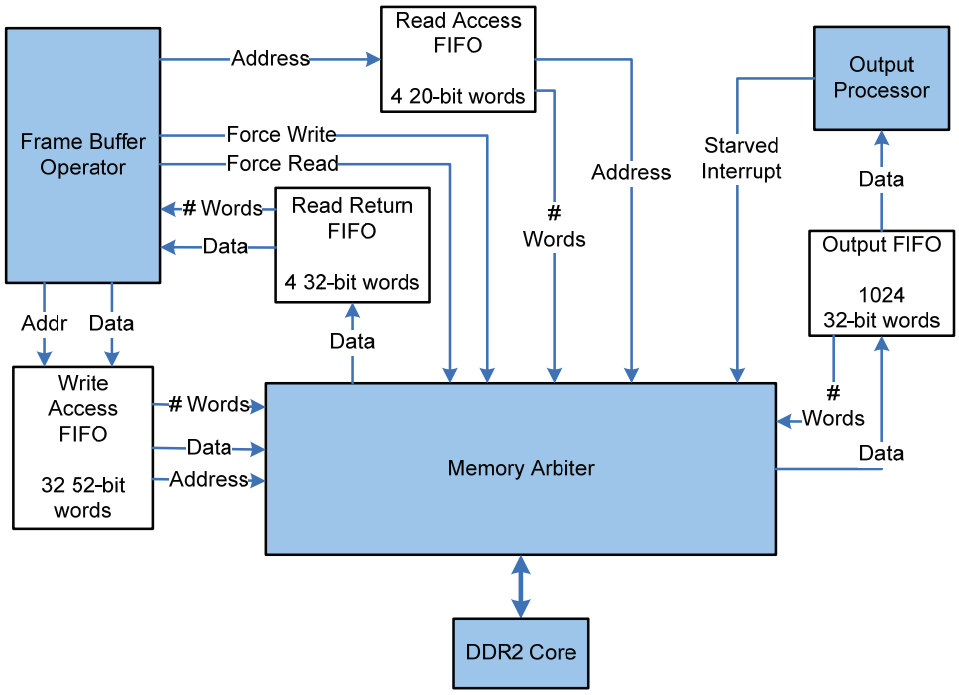


Figure 32: Memory interconnect diagram.

The read access FIFO stores memory addresses for FBO read transactions, and the read return FIFO stores the resulting data from the memory. The write access FIFO collects writes from the FBO until the MA can execute the write transactions. These three FIFOs provide buffering that allows the FBO to continue operating with minimal impact from the DDR2 controller's memory latency.

On the output side, another FIFO buffers the data between the MA and the Output Processor (OP). In normal operation, the MA fills this FIFO whenever it drops below half full. However, if the FBO needs access to the memory as well, there is a potential for conflict. The priority-based memory arbitration scheme allows maximum bandwidth for the FBO while still allowing the OP to retrieve each video frame. The highest priority is an OP starvation, which means that less than 200 pixels are remaining in the output

FIFO. Next on the priority list are writes and reads from the FBO and then a standard read transaction to fill up the output FIFO. The burst capability of the DDR2 memory interface allows the output FIFO to recover very quickly (at 32 pixels every 25 clocks) from a starvation so that the FBO can regain access to the memory. The lowest priority in the arbitration scheme is the PCI interface since it is only used during diagnostic activities. Most of the traffic from the PCI interface enters the graphics pipeline instead of drawing pixels directly in the frame buffer.

In addition to arbitrating between the FBO and the Output Processor, the MA also converts the 64-bit DDR2 data path into the individual 32-bit pixel data values. Because the DDR2 controller is operating in half-rate mode (150 MHz), the data width is doubled twice – once for the half-rate mode and once for the double-data rate. Thus, the local data width is 256 bits, or eight pixels. The MA retrieves each block of eight pixels and stores them in the mixed-width Output FIFO. The output side of this FIFO has a 32-bit width operating at the pixel clock frequency, which is based upon the display resolution and refresh rate. This mixed-width dual-clock FIFO is another example of how this architecture has been crafted specifically to take advantage of the FPGA's available structures.

Unlike the sequential memory addresses from the OP, the FBO typically provides a much more random pattern of pixel addresses to the MA. To maximize the efficiency of the memory bandwidth, the MA compares the upper bits of the pixel addresses to collect as many write transactions as possible into the burst of four words (8 pixels each). The maximum number of pixels written in one burst is 32 pixels, and this maximum

efficiency occurs when the pixels are written in order – such as during the operation that clears the frame buffer at the beginning of each frame.

3.2.5 Output Processor

The Output Processor is responsible for retrieving pixels from the front frame buffer and transmitting them out according to the timing of the VESA specifications. The OP is parameterized to provide three different resolutions: VGA, SVGA, and XGA. Because of the different pixels clocks required for the different resolutions, the OP has been segregated into its own clock domain, with the actual clock frequency based upon the resolution. Table 3 shows the pixel clocks required for the various resolutions. The Output Processor uses a PLL with an input reference clock of 150 MHz, which is the base clock for the rest of the architecture. From this reference frequency, the PLL is parameterized to accept different register values that are used to generate the output clock frequency. The PLL is reconfigured by shifting in new settings via the PLL's serial shift-register chain, as described in Altera's PLL Reconfiguration application note [73]. A detailed RTL diagram of the Output Processor is shown in Appendix G.

Table 3. Pixel clock frequency based on resolution.

Resolution	Horizontal Pixels	Vertical Pixels	Total Pixels	Bandwidth (Mbps)	Pixel Clock (MHz)
VGA	640	480	307200	589.8	25.25
SVGA	800	600	480000	921.6	40.00
XGA	1024	768	786432	1509.9	68.00
SXGA	1280	1024	1310720	2516.6	115.00

The use of a reconfigurable PLL is another example of how this graphics processor architecture was crafted specifically for an FPGA to take advantage of its available resources. Modern FPGAs often have between one and four PLLs designed into the die as dedicated silicon. Each PLL consists of a phase-frequency detector, charge pump, voltage-controlled oscillator, and multiple counters. None of these features actually consume any logic elements as part of the programmable portion of the fabric.

3.2.6 Scalability

Overall, the basic graphics processing provided by this architecture provides for a broad impact because it is advantageous in a variety of applications that require 2D graphical overlays, menu systems, or even flight-critical avionics symbology. Almost as important, though, is the fact that this architecture is built for scalability and extendibility to address even more applications with only minor additional effort.

To improve the overall graphics performance and fill rate, the general approach is to move more functionality into parallel processing paths. There are two different ways in which this architecture can be parallelized: by making each block more parallel, or by replicating the entire pipeline. In fact, the basic graphics pipeline described here can be thought of as just one tile in a multi-tiled GPU. Careful consideration has to be given to the input and output of each pipeline tile, but overall the performance increases greatly without having to redesign the intricate details of the graphics processing. The input to each pipeline tile can be divided up based upon screen area, primitive type, or just in a round-robin fashion. The output from each pipeline tile can be buffered in FIFOs and

then fed into a single frame buffer manager which would arbitrate between the pipeline tiles. In addition, the frame buffer memory interface can be widened to reduce bottlenecks due to multiple pipelines.

As an alternative to creating multiple pipelines, the architecture can also be extended simply by developing additional blocks to provide more capabilities. This extendibility is demonstrated in Chapter 6.

3.3 FPGA Resources

This architecture was developed to provide basic graphics processing capabilities while requiring minimal FPGA resources to allow for expandability. The research was conducted in an Altera Stratix III development kit, which includes an EP3SL150 device. The EP3SL150 device has 56,800 adaptive logic modules (ALMs), 5.5 Mbits of embedded memory, and 384 DSP blocks [13]. Each ALM has eight inputs and is capable of performing several different functions based upon its configuration. This graphics processing architecture uses about 25% of the available ALMs and about 65% of the embedded memory, as shown in Table 4. The design also uses only 48 DSP blocks to perform the required multiplication.

Table 4. FPGA resource summary.

Module	ALMs	Block Memory Bits	M9Ks	M144Ks	DSPs
Nios	2020	1262088	35	8	4
Rasterizer	5654	1074688	143	0	12
Frame Buffer Operator	1061	851968	8	6	32
Output Processor	1041	65770	9	0	0
Memory Arbiter	1618	53248	6	0	0
DDR2 Controller	2748	512	1	0	0
TOTAL	**14398**	**3545842**	**231**	**14**	**48**
AVAILABLE	56800	5499000	355	16	384
PERCENT USED	25.3%	64.5%	65.1%	87.5%	12.5%

CHAPTER 4

SIMULATION APPROACH & RESULTS

4.1 Overview

The analysis and performance assessment of the FPGA GPU architecture was first conducted in a simulation environment. This environment was constructed based upon several needs including the ability to change input commands and data easily, analyze individual pixel data, and visually inspect the output. Several new tools were developed to help create this simulation environment that provides quick visual results for analysis.

Simulation was more than just a verification task that was completed at the end of the architectural development. Instead, simulation was relied upon from the very beginning of the research and was utilized at all levels of the design hierarchy. The results of the lower level simulations drove design decisions that impacted the overall GPU architecture. The results of the final end-to-end simulation validated some architectural constructs while also revealing some shortcomings and changes needed in the overall architecture. The remainder of this chapter first discusses the simulation approach and then analyzes the simulation results.

4.2 Simulation Approach

The simulation approach uses a combination of standard tools and custom-developed utilities and scripts to provide a fully automated visual output based on the

specified stimulus [74]. The environment is shown Figure 33 as a block diagram. The blue shaded blocks were developed specifically for this architectural research, and the boxes with dashed lines are human-readable files for entry or analysis.

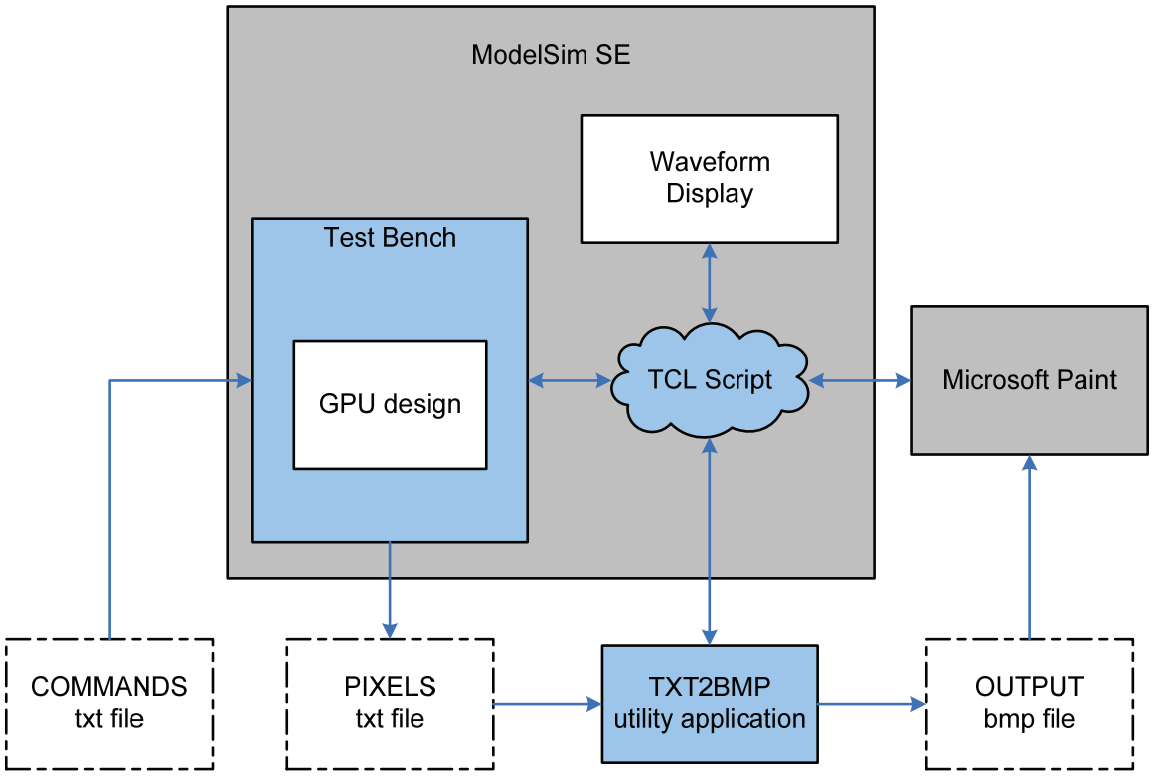


Figure 33: Environment for simulation approach.

ModelSim SE with the Mixed-HDL feature was used as the basis for the FPGA simulation. All of the code was designed in VHDL, but the Mixed-HDL feature was needed because the memory models and the Rasterizer's multi-threaded processor were both provided in Verilog. The test bench forms a wrapper around the GPU design, much like any other simulation approach, but it also controls file input and output. The TXT2BMP utility was developed to convert the output pixel text file into a bitmap that

can be viewed in Microsoft Paint. A custom TCL script provides the automation that ties together all of the utilities and files. Each of these elements is described in more detail in the following sections.

Although only one environment is shown in the figure, simulation was used at nearly every level of hierarchy during the architectural development. Each module in the architecture has its own test bench with unique stimulus and interconnect. As the performance of each module was validated, the test bench of the next higher module was developed and used for simulation. This process continued until the final end-to-end simulation of the entire architecture was accomplished.

4.2.1 Test Bench

A typical test bench acts as a wrapper around the design under test and provides interconnect and stimulus for the design [75]. In the VHDL hierarchy, the test bench is one level above the design under test, which is treated as a component in the test bench. During the development of the test bench, there were three increasing levels of complexity:

1. Simple stimulus coded in the test bench and results viewed as waveform
2. Simple stimulus but results viewed in output text file
3. Stimulus loaded from external text file and results viewed in output text file

To validate many of the lower level modules in the architecture, the test benches were initially very straightforward with the results viewed as waveforms. This low level analysis ensured that the timing of state machines and interfaces was correct and resulted in the desired functionality. As the architecture was further developed, it became

necessary to run the simulation with a number of different command inputs and data inputs. This exhaustive simulation was necessary to ensure the architecture's robustness but also to provide a comprehensive set of results for performance analysis. The large amount of simulation required led to the need to automate the simulation with text file inputs and eventually bitmap file outputs.

The final test bench uses a large while loop to cycle through each line in the command stimulus file, the interface for which was developed by another researcher at L–3 Communications. Figure 34 shows the structure of a sample stimulus file. The first word in each line denotes the type of command, and each command can have up to four arguments. All stimulus files have to end with the "FINISH" command in order to trigger the test bench to stop the simulation and process the output results. The trailing lines at the end of the command names are necessary for the VHDL procedure to parse the text commands correctly. The sample file below enables anti-aliasing and then draws a triangle with vertices at (500, 500, 9), (550, 500, 0), and (500, 550, 9). In this particular example, the last two vertices in the file are ignored by the GPU because a third vertex is not present.

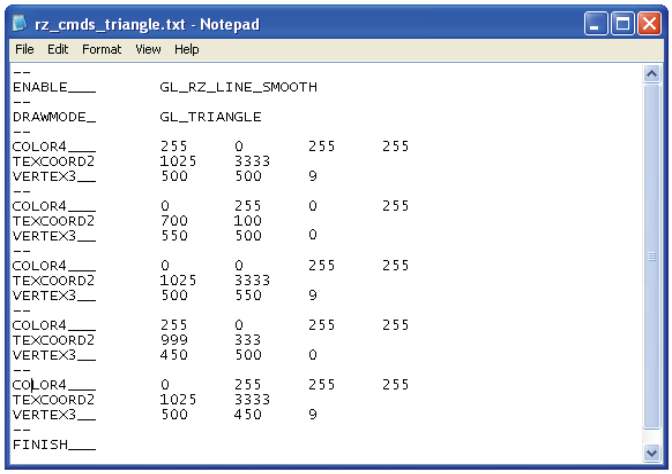


Figure 34: Sample input command text file.

A separate process in the test bench detects output pixels and stores them in a text file. This text file of pixel data can either be used for manual analysis or as an input to the TXT2BMP utility that is described in the next section. The structure of the output pixel text file is shown in Figure 35. The structure actually changed quite a bit through the course of development. The sample file shown in the figure has a 16-bit X value, 16-bit Y value, and a 32-bit RGBA value to describe each pixel. This data vector was tapped off the output from the Rasterizer in the architecture. When validating the output of the Frame Buffer Operator, the file structure changed to include a 32-bit memory address instead of the X and Y coordinates.

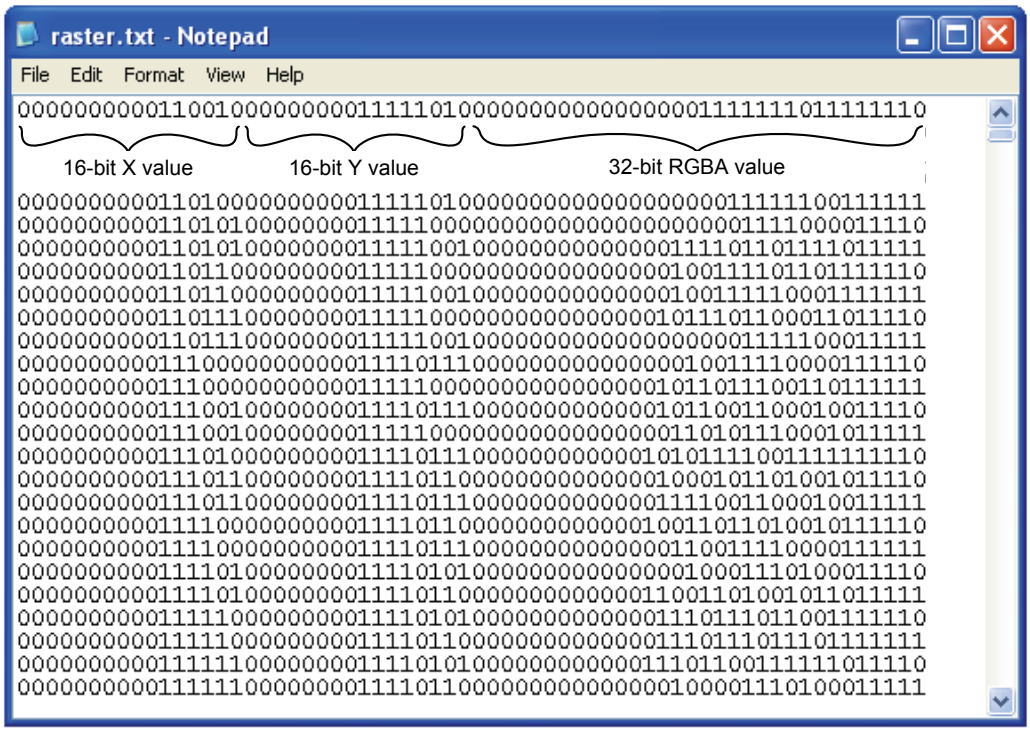


Figure 35: Structure of output pixel text file.

4.2.2 TXT2BMP Utility

The TXT2BMP utility software was developed to convert the simulation's output pixel text file into a visual representation that can be analyzed much quicker than the tedious process of manually converting and drawing pixel data. The software is driven from a command line and has several options available as shown in the following table. The TXT2BMP utility can also convert data received from a bitmap capture operation with the actual hardware. The source code for TXT2BMP is included in Appendix H.

Table 5. TXT2BMP features.

Options	Description
-r | -f | -b	r = rasterizer format (X + Y + RGBA) f = frame buffer format (ADDRESS + RGBA) b = bitmap format (sequential RGBA values comma-delimited)
-x | -v	x = XGA resolution v = VGA resolution
-b | -w	b = black background w = white background
-l | -p	l = landscape orientation p = portrait orientation

The TXT2BMP utility works by iterating through each line in the input file and creating a new bitmap file with modified pixel entries. To simplify the utility software, a set of eight blank bitmap files was created and then used as a reference in the software. The eight bitmaps comprise every combination of the resolution, background, and orientation options. These files are used as a starting point in the utility so that the BMP file header structure did not have to be recreated each time and the unwritten pixels (background) also did not have to be computed. The actual pixel data in the bitmap file structure begins at a memory offset of 0x36. Each pixel receives three bytes of data

(RGB) and is ordered from the bottom left corner, going from left to right, and then incrementing rows until you reach the top of the image [76]. The pixel location is translated from either the XY or memory address into the memory offset required for the bitmap file.

4.2.3 TCL Script

A TCL script was developed to automate the simulation and accelerate the analysis time. The script, shown in Figure 36, first recompiles the test bench (and its dependent package) to pull in any changes that were made since the last simulation. The script then initiates the actual simulation environment in ModelSim with the "vsim" command. Since Verilog does not allow explicit library declarations, multiple arguments are used after the "vsim" command to link in the correct libraries. The final two lines in the script setup the waveform window with predefined signal formats and then run the simulation indefinitely.

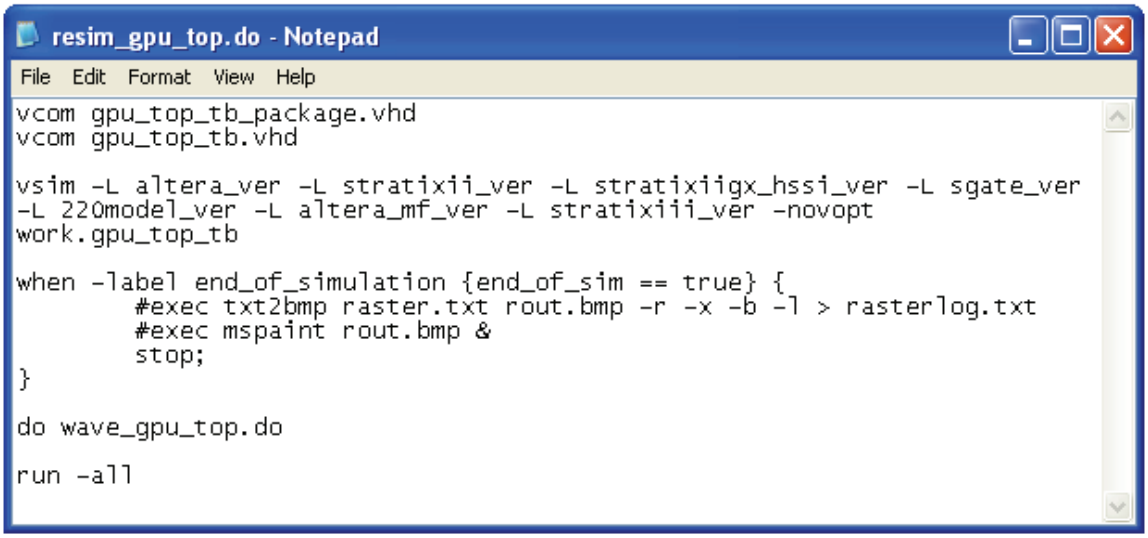


Figure 36: Test bench source code.

As previously described, the test bench requires the stimulus file to include the "FINISH" command as the final entry. When this command flushes through the GPU pipeline, the test bench triggers an "end_of_sim" variable. This "end_of_sim" variable informs the TCL script that it is time to invoke the TXT2BMP utility and then to open Microsoft Paint with the newly converted bitmap output file. Once both of these applications have been executed, the simulation stops and is ready for analysis in the waveform window, text file, or bitmap file.

4.3 Simulation Results

This simulation results helped drive architectural design decisions and also validated the performance calculations. Numerous command stimulus files were used during the simulation to verify the correct rendering of points, lines, and triangles in various orientations, shapes, sizes, and quantities. Many rasterization bugs and artifacts were discovered during the course of the simulation. For example, the triangle shown in Figure 37 appears to be nicely blended with anti-aliased edges. However, upon zooming in to the top corner of the triangle, there is a one-pixel artifact as shown in Figure 38. This pixel was the result of the Rasterizer drawing the triangle edges but not generating a horizontal line fill between the edges. Because the required horizontal line was only one pixel long, the algorithm failed to draw it. The visual nature of the simulation results allowed for both a quick identification and solution to the problem.

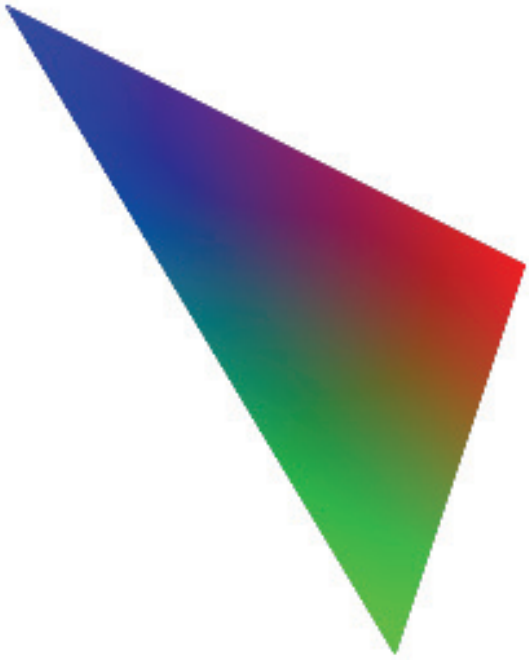


Figure 37: TXT2BMP output of blended triangle.

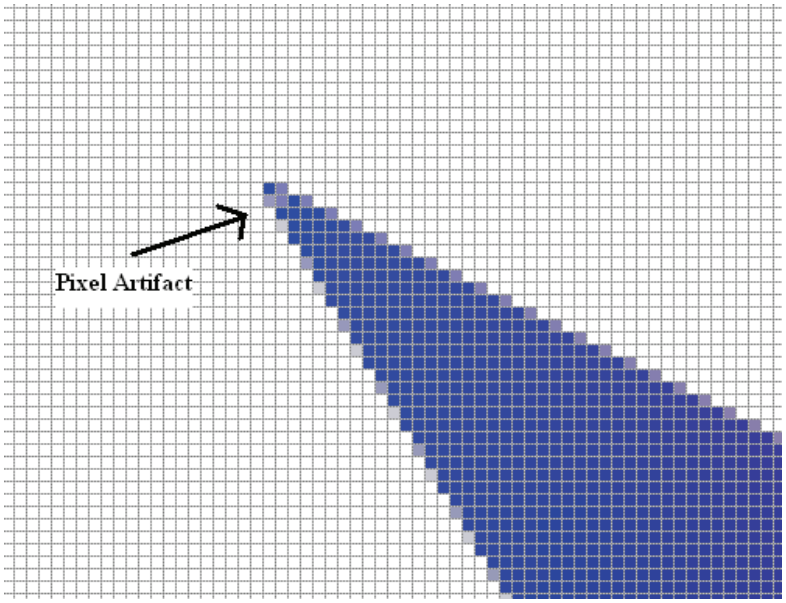


Figure 38: Highly magnified TXT2BMP output.

The waveform display in ModelSim was also used to detect timing or sequencing errors in state machines. An example of the waveform display is shown in Figure 39. This figure shows the Rasterizer's setup module calculating the gradients for a line. The process begins when the data is received on the event interface (eif) and then concludes when the results are transmitted out on the task interface (tsk).

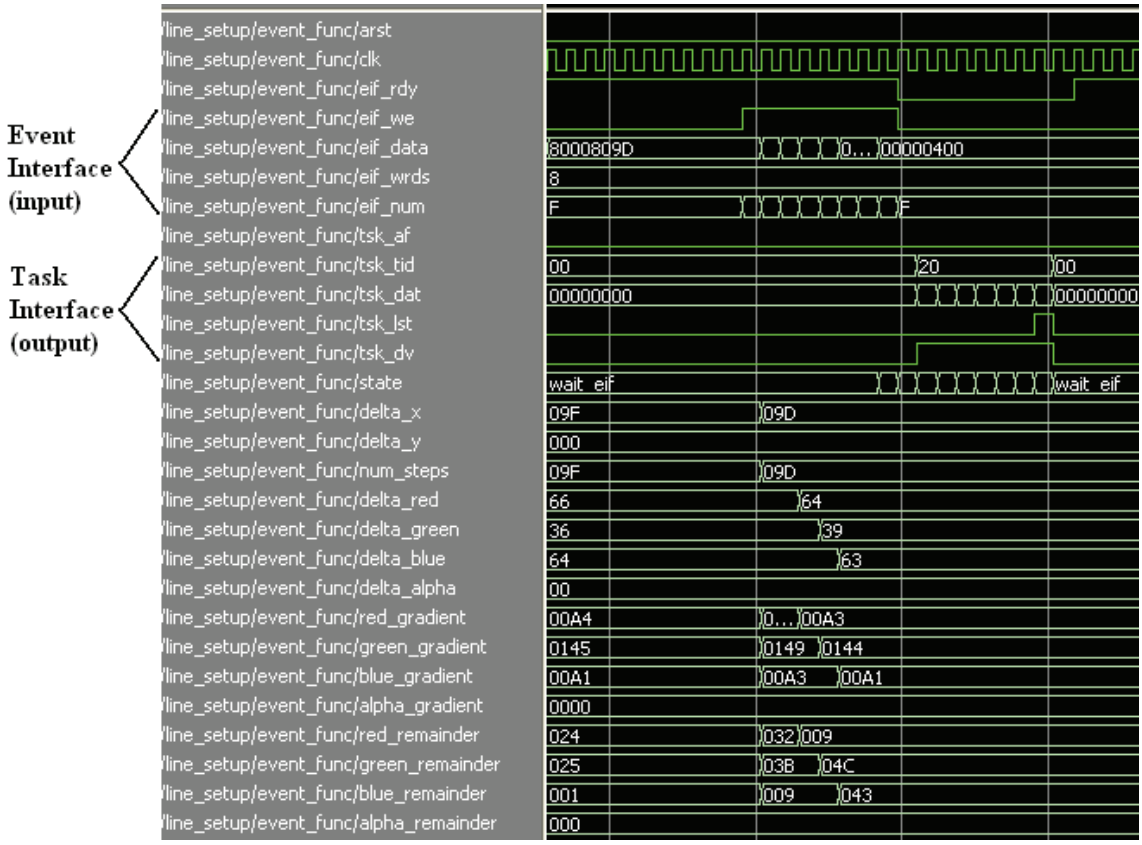


Figure 39: Waveform results of gradient calculations.

Overall, the simulated results match the expected results based upon the calculations performed during the architectural development. The performance results from the simulation are summarized in the table below. The statistics from the host

interface were not included because they are heavily dependent on system-level traffic and processing.

Table 6. Simulation results.

Functionality	Clock Cycles	Comments
Line Setup	28	Includes slope and gradient calculations
Fragment Generation (Anti-aliasing on)	2.5	Each pair of anti-aliased fragments requires 5 clock cycles
Fragment Generation (Anti-aliasing off)	2	
Fragment Scissoring	1	
Fragment Alpha Blending (Pixel Cache hit)	2	Assumes no read from DDR2 memory is necessary
Fragment Alpha Blending (Pixel Cache miss, burst of four fragments)	7	Assumes burst of 4 fragments being blended at same time; Limited by the 21-clock cycle read latency of DDR2 interface
Fragment Alpha Blending (Pixel Cache miss, single fragment)	25	Assumes a single fragment is being blended; Limited by the 21-clock cycle read latency of DDR2 interface
Fragment Write to Frame Buffer (Consecutive addresses)	18 clocks for 32 pixels	Assumes all 32 pixel addresses are consecutive and within one row; Limited by the 13-clock cycle write latency of DDR2 interface
Fragment Write to Frame Buffer (Single pixel address)	14	Assumes a single fragment is being written; Limited by the 13-clock cycle write latency of DDR2 interface;

The simulation results show that each line requires 28 clock cycles for setup with either an additional two clocks per aliased pixel or 2.5 clocks per anti-aliased pixel in the line. The performance of the Frame Buffer Operator depends upon the pixel cache results as well as the DDR2 memory interface latency. In the best scenario (pixel cache hit), the alpha blending requires two clock cycles per pixel and then the frame buffer write requires 18 clock cycles for 32 pixels. For the worst case scenario (pixel cache miss), a single pixel can consume up to 25 clocks for alpha blending and 14 clocks for frame buffer writing, for a total of 39 clock cycles.

Based on the simulation results, several changes were made to the FPGA GPU architecture to avoid the worst case scenario. The triangle filling algorithm was altered to use horizontal lines to fill the interior. Horizontal lines are the most efficient because they are comprised of pixels with successive memory addresses, and this consecutiveness takes advantage of the DDR2 memory's burst capabilities to write 32 pixels in 18 clocks. The architecture was also optimized by increasing the pipelining within each of the individual modules along the overall GPU pipeline. For example, the Frame Buffer Operator can simultaneously operate on several fragments in various stages such as scissoring, alpha blending, and frame buffer writing. Figure 40 shows that due to pipelining both the Rasterizer and the Frame Buffer Operator, three lines can be in-process at the same time. This pipelining significantly improves the performance over traditional batch processing but was not realized until end-to-end simulations were conducted.

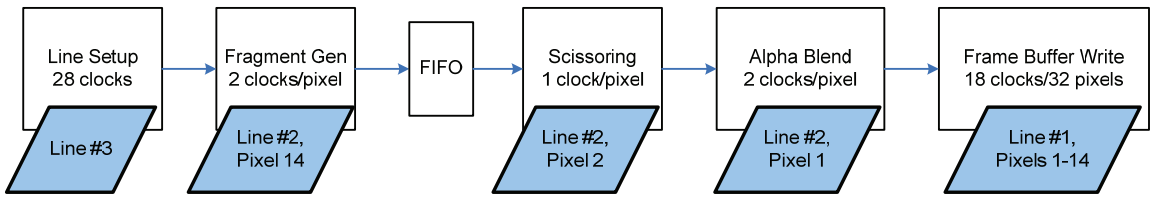


Figure 40: Example of pipelining optimizations.

The simulation results can be extrapolated to calculate broad GPU performance metrics. The actual performance is highly dependent upon the stimulus to the GPU as well as the memory interface latencies. If no alpha blending is assumed, then each pipeline stage in the architecture requires fewer clock cycles than the previous stage. As

such, the performance will be throttled by the speed of the Rasterizer at the beginning of the pipeline. With a 150MHz clock frequency, the FPGA GPU architecture can process over 10.7 million vertices per second. In addition, 75 million aliased fragments or 60 million anti-aliased fragments can be generated. These extrapolations assume that the pipeline does not stall because of memory latencies or pixel cache misses during the alpha blending. The actual delays due to the memory interface are dependent on the application, so the metrics presented are the upper bound of performance.

There are 307200 pixels in a VGA resolution display, and with a refresh rate of 60 Hz, there are 18.4 million pixels per second. For an XGA resolution display, there are 46.7 million pixels per second. Thus, the simulation results reveal that the FPGA GPU architecture can render four times the number of VGA pixels/sec and 1.6 times the number of XGA pixels/sec. These ratios demonstrate that the architecture has ample performance to satisfy a variety of applications.

CHAPTER 5

AVIONICS APPLICATION

5.1 Introduction

The avionics industry is an appropriate market for the application of the FPGA GPU architecture because it has been challenged for years to maintain cost effectiveness while introducing new visually appealing functionality such as data fusion, synthetic vision, and 3D maps. COTS graphics processing units (GPUs) are often relied upon to satisfy the high-performance requirements, but these devices were not originally designed for the avionics industry. The avionics environment imposes temperature, power, lifecycle, and certification constraints that are not addressed by COTS GPUs. The certification of GPUs has become a particular issue in recent years with the release of the Certification Authorities Software Team (CAST) Position Paper 29, "Use of COTS GPs in Airborne Display Systems" [77]. Because of the complexity and criticality of GPUs in avionics, the CAST believes that COTS GPUs require special attention.

The major markets driving COTS GPU development have typically been PC gaming and then more recently portable and automotive electronics. The leading GPU developers, ATI (now a part of AMD) and Nvidia, do not continue to produce or even readily support the GPUs that they have released in previous years. In avionics, however, long-term supportability is a key component of lowering life cycle cost. The nearly continual gaming GPU product release cycle means that each device is only available for

a very limited time – sometimes only six to nine months, which certainly does not support the years or decades of life cycle support required in avionics. These gaming GPUs also offer much more functionality than is required by avionics, which is usually interested in just a small subset of OpenGL.

In the portable and automotive markets, the primary trend is towards GPU cores that are integrated into a custom System on a Chip (SOC). SOCs are extremely cost efficient when the product volumes are very high, which is not usually the case for avionics applications. In addition, some of the performance requirements in avionics for demanding applications such as synthetic vision may be too intensive for the GPU cores that were designed for cell phone and other low resolution applications.

Avionics graphics processing lies somewhere between high-end gaming GPUs and the low-end but readily available SOC GPU cores. If the graphics processing were instead hosted in a field-programmable gate array (FPGA), the performance could be customized for the avionics industry without an immediate fear of obsolescence. This chapter first provides conclusive motivation for an FPGA-based GPU in avionics applications. Then an implementation of the previously described FPGA GPU architecture is explained with the results analyzed to ensure that the architecture satisfies the avionics requirements.

5.2 Graphics Processing in Avionics

The avionics industry imposes a unique set of requirements for graphics processing. These requirements encompass not only graphics functionality but also environmental, life cycle, and certification constraints. Each of these requirements is

discussed in the following paragraphs. This section concludes with a summary of the GPU options that have been researched for avionics applications [78].

5.2.1 Graphical Functionality

As cockpits have collected more advanced sensors and computational capabilities over the past decade, the requirement to display more advanced graphics has also increased. The pure amount of information available to the pilot today requires careful attention to human factors issues in order to maximize situational awareness. Captain Runyon very aptly describes the information available in a prior generation: "In the early years of military aviation, the pilot obtained sufficient tactical information by simply looking out the cockpit window" [79]. Now that much more data is available to the pilots, the burden is upon the application developer to utilize the graphics processor's advanced features to consolidate and simplify the presentation of the data.

When LCDs first started to penetrate the cockpit, they most often replaced a single mechanical instrument or perhaps a small cluster of gauges. However, now large-area multi-function displays are prevalent because of their ability to provide many different types of information as well as their ability to display new data formats/videos as they become available. The fifth-generator fighter aircraft in development today, the F-35 Joint Strike Fighter from Lockheed Martin, includes a 20" x 8" LCD with full windowing capabilities to display aircraft, mission, and navigational information all on one display.

Just as more recent LCD-based cockpits have migrated towards large area displays, the graphics processing horsepower required for those displays has also increased. From only a memory bandwidth perspective, the performance requirements

have increased over 10x – from just 20.7 MB/sec in a 30 Hz 6-bit color VGA display up to 235.9 MB/sec in a 60 Hz 8-bit SXGA display. Figure 41 shows the F-35 Panoramic Cockpit Display with a double-wide SXGA resolution [80].

Figure 41: Panoramic cockpit display.

In addition to the memory bandwidth increases, the imagery on the displays has also become much more realistic in order to improve human factors issues. Technology such as synthetic vision, highway-in-the-sky, and electronic "steam" gauges all attempt to provide more information to the pilot with less confusion. Figures 42 and 43 show various display formats of the SmartDeck™ primary flight display (PFD) and multi-function display (MFD) [81].

Figure 42: Synthetic Vision on PFD.

Figure 43: Engine gauges on MFD.

5.2.2 Environmental Considerations

The overall push for cost effectiveness in avionics has led to an increased use of COTS devices. These devices are often not rated for the wider temperature ranges (−55°C to +71°C) that can be seen in military avionics. To justify the use of commercial-grade parts in industrial-grade applications, the components can be de-rated such that they are not subjected to the maximum stresses of their full ratings. For example, a capacitor rated to 50V should only be used in a 25V application if the temperature range is extreme. This type of de-rating not only helps the device work over larger temperature ranges, but it also improves long-term reliability.

De-rating graphics processors generally results in a reduction of the clock frequency. When doing this, the overall performance of the graphics processor decreases in a linear fashion. For a clock running at half the rated frequency, the GPU will provide half the throughput. This performance impact may prove detrimental when trying to accomplish the advanced rendering described in the previous section.

5.2.3 Certification Requirements

For many years, testing avionics based upon well-defined functional requirements at the product level was often sufficient enough to deem products worthy of flight. However, as both design complexity and safety criticality awareness have increased, the FAA has issued guidelines that stipulate a much more formalized certification process. In the early 1980s, the FAA issued DO-178, Software Considerations in Airborne Systems and Equipment Certification [82]. DO-178 has been subsequently revised in 1985 and

1992, with revision C due out possibly late in 2011. The DO-178 guidelines dictate a formalized development process and a requirements-based design.

The hardware within avionics has largely been excluded from a formal certification process until just recently with the introduction of DO-254, Design Assurance Guidelines for Airborne Electronic Hardware [83]. DO-254 also now encourages a requirements-based design with full code coverage through simulation and hardware testing. One of the major issues with DO-254 is that FPGA designs often depend upon COTS IP cores that do not have clear source code available. The source code is required for safety criticality levels A and B because of the need for elemental analysis, which means that every line of code must be exercised. If COTS IP cores are simply treated as black boxes and just functionally tested, the complete scope of DO-254 is not satisfied.

In addition to the burden of DO-254, a CAST position paper (CAST-29) also specifically targets COTS graphics processors [77]. Their extra attention is due to the fact that the COTS GPUs represent a safety risk as more graphics functionality is offloaded from the DO-178 certified software into high-speed hardware devices that were designed for an entirely different market – video games and computer graphics. The CAST position paper identifies eight concerns or issues with COTS graphics processors:

1. DO-254 assumes that COTS components will be verified at a system level, but COTS GPUs are too complex for this approach.

2. COTS GPUs can contribute to hazardously misleading information (HMI) because of hardware failures, design errors, or inappropriate responses to environmental conditions.

3. COTS GPUs that are used in multiple, redundant displays do not provide risk mitigation because of the possibility of generic design errors.

4. Performance variations may exist across the production lifetime of the COTS GPU devices.

5. COTS GPUs may have configurable elements that introduce the possibility for an uncontrolled configuration of the hardware.

6. COTS GPUs may undergo die revisions or process changes without changing the part number of the devices.

7. COTS GPUs often have a large amount of unused functionality in avionics applications, and a failure could produce unintended operation of the device.

8. COTS GPUs require OpenGL software drivers, and the developers of those drivers are sometimes not cognizant of DO-178 certification requirements.

5.2.4 Life Cycle Constraints

The life cycle of a typical consumer product could be years or even just months, but avionics equipment is expected to be in service or at least maintainable for possibly decades. Therefore, when avionics equipment is too heavily based upon consumer technology, the life cycle support becomes a tremendous challenge. In addition, the design and certification of avionics equipment can be a costly endeavor, so there is additional motivation to design products that can survive many years of production. Sometimes a technology refresh can be used to replace obsolete components, but overall the life cycle cost of avionics is directly tied to reliability and obsolescence. The longer

that the individual components are available, the less reinvestment required for that product. Two patents were awarded based upon this work to develop new replacement avionics instruments with longer component availabilities compared to the existing instruments [84, 85].

5.2.5 Options for GPUs in Avionics

There are five basic approaches for implementing graphics processing in any application: a discrete GPU, a CPU with embedded GPU, a software GPU, a custom GPU ASIC, or an FPGA. The advantages and disadvantages of using each of these in avionics will be discussed in the following paragraphs.

Discrete GPUs

Discrete GPUs, such as the latest chips released from AMD or Nvidia, provide the most powerful graphics processing available. Table 7 shows a list of commonly used discrete GPUs that have DO-178 certified drivers. The three most popular AMD GPUs that are used in avionics are the M9 (Mobility Radeon™ 9000), M54 (Mobility Radeon™ X1400), and most recently, the M96 (Radeon™ E4690). The M9 and M54 were originally targeted for laptop applications as they require a bit less power than the traditional workstation GPUs. The M96, on the other hand, was developed for embedded applications and AMD currently projects a 5-year lifespan for the device [86]. ALT Software, a DO-178 device driver development company, has also partnered with

Channel One International to guarantee a 20-year component availability of the M96. This type of long-term agreement should mitigate some of the risk of obsolescence [87].

The Fujitsu GPUs were all developed for the embedded automotive market, which requires graphics overlay blending with external video sources [88]. This similarity with many avionics requirements makes the Fujitsu GPUs a good fit for some applications, but the GPU still raises certification concerns in safety-critical applications because of its complexity and lack of formal verification.

Table 7. Discrete GPUs.

GPU	Released	Host I/F	DO-178 Driver	Pipeline	Clock (MHz)	Memory I/F
AMD M9	2002	PCI	ALT	Fixed	250	DDR
Fujitsu Coral PA	2005	PCI	ALT	Fixed	166	SDRAM
Fujitsu Carmine	2005	PCI	ALT	Fixed	266	DDR
AMD M54	2006	PCIe	ALT	Shader	445	DDR
Nvidia G73	2006	PCIe	Presagis	Shader	500	DDR2
Fujitsu Ruby	2009	PCIe	ALT	Shader	266	DDR2
AMD M96	2009	PCIe	ALT	shader	550	DDR3

Presagis (formerly Seaweed Systems) has partnered with Nvidia to develop a DO-178 device driver for the G73 (GeForce 7300) GPU. This GPU was designed for high-end gaming applications so its performance is certainly satisfactory for avionics, but the long-term device availability is still an issue [89].

None of the devices listed in the table were designed for the certification, performance, and life cycle requirements of avionics. As a result, choosing any of the discrete GPUs will result in a compromised solution.

CPUs with Embedded GPU

A CPU with an embedded GPU, such as the PowerVR core from Imagination Technologies, is essentially a System-on-Chip approach. Imagination Technologies does not manufacture any GPU devices, but instead they license their PowerVR cores to microprocessor companies such as Intel, Freescale, and Texas Instruments. There are two main PowerVR cores – MBX and SGX. MBX is targeted for the handheld or portable market and uses a fixed-function pipeline. SGX is targeted for higher performance with a fully programmable pipeline, and it is used in higher-end devices such as the Apple iPad and other netbook applications [90].

There are two approaches to using the PowerVR GPU cores in avionics – the core could be licensed and integrated into a new custom ASIC, or one of the existing COTS SoC devices could be used. For medium to small-volume applications such as avionics, the return on the NRE investment for a custom SoC will likely not be beneficial. However, an existing SoC device could be used without any NRE investment. Each of these devices, such as the TI OMAP or Intel ATOM, is already functional and has the CPU tightly integrated with the GPU. This existing integration reduces the development risk of a new product, but there are also disadvantages for avionics applications. The performance is not scalable from low-end applications up to very demanding applications. With an SoC, the performance is typically just as advertised on the datasheet. If the application has growth features that require a higher performance, then a more powerful SoC has to be designed in from the beginning. This design decision ripples into increased certification costs and increased power.

Another disadvantage in using SoC GPUs is that the PowerVR GPU cores were designed for completely different applications – handheld devices and netbooks. When the cores were designed, several assumptions had to be made regarding the ratio of pixels to texels (texture elements). This assumption could have driven design decisions that resulted in a misallocation of the performance optimization for non-targeted applications. For example, the PowerVR cores were optimized to accelerate rasterization through tile-based rendering. There is very little focus on the texture and shading processor [91]. An avionics application that displays weather RADAR and multiple videos can typically use upwards of four or five large-area textures for each frame. This specific performance requirement results in a high demand for texture memory bandwidth – an un-optimized portion of the PowerVR design.

Software GPU

Quantum3D has developed IGL 178, which is a software-based GPU [92]. This GPU implements all the OpenGL SC functionality entirely in software, with no hardware acceleration. The software GPU (estimated to be less than 15,000 lines of code) is extremely portable because it can be hosted on any microprocessor. As such, this approach very aptly addresses the certification and device life cycle concerns, but it is limiting when high graphics performance is required. One benchmark compared the software GPU to a hardware GPU, and the software GPU rendered the graphics at half the rate of the hardware GPU [93]. Performance optimizations such as FPGA-based accelerations or parallel processing could improve the frame rate enough for some more advanced graphics applications.

Custom GPU ASIC

A custom GPU ASIC could be developed either based on existing GPU cores (such as PowerVR) or designed based upon the specific application's requirements. With either approach, a custom GPU ASIC has two distinct disadvantages in the avionics market. First, the NRE cost for ASIC development is difficult to justify unless there is a large quantity of GPUs that would be produced. In avionics applications, the quantities are typically several thousand units or less – not millions of units. These low quantities lead to a poor return on the initial NRE investment.

The second disadvantage of a custom GPU ASIC for avionics is that the high NRE forces the designer to broaden the scope of requirements instead of optimizing for a particular application. This approach could lead to a GPU that almost works satisfactorily in many different applications but does not exceed performance requirements for the target application. Alternatively, the GPU could be overdesigned for the target application and as a result suffer from power, cost, and certification increases.

FPGA-Based GPU

An FPGA-based GPU provides life cycle, certification, and flexibility advantages for avionics applications. Because FPGAs are not closely tied to consumer applications, they are usually available for at least 10-15 years instead of sometimes less than a year for discrete GPUs. Furthermore, if the FPGA device does go obsolete, the GPU design could be retargeted to an alternate FPGA with relatively little design effort and minimal certification effort.

A GPU that is hosted in an FPGA mollifies the concerns listed in the CAST position paper because it allows full DO-254 certification from the clear source code.

This straightforward path to certification provides risk mitigation for both cost and schedule impacts on development programs.

The GPU performance required in avionics varies widely from each application to the next. Some displays require only simple graphics with menus, but other displays require full 3D rendering with texture blending and high resolutions. The wide range of GPU performance in avionics makes the choice of an FPGA-based GPU very appropriate. The FPGA-based design can be re-programmed with a modified GPU pipeline that focuses its performance optimizations just in the areas needed. These application-based optimizations result in a power-efficient design.

The two most notable disadvantages to an FPGA-based GPU are the initial development costs as well as the limited performance capabilities. The initial NRE for the development of the core technology is substantial, but it is less than the NRE required for custom ASIC developments. The FPGA-based GPU's performance is also only as good as its firmware design. An FPGA-based GPU is certainly not going to match the performance of the latest AMD or Nvidia GPUs, but the avionics requirements are also much less than the gaming applications that drove the design of those discrete GPUs.

5.3 FPGA-Based GPU Replacement

5.3.1 Goal

The Displays Systems (DS) division of L-3 Communications designs, integrates, and manufactures mainly military cockpit displays. Each cockpit display can be thought of as a mini-computer complete with a CPU, GPU, I/O card, power supply, display, and

user interface. The GLINT GPU used in the SuperSet™ product family was discontinued by the manufacturer in 1999, but L-3 DS has kept the SuperSet™ products in production by purchasing excess components from trusted component brokers and other third parties. However, now the GPUs and their associated video RAM (VRAM) are very difficult to find. In order to overcome this obstacle as well as to provide for a long-term solution to the GPU obsolescence, L-3 DS desired to have a permanent solution for the GPU functionality. This solution would need to be a drop-in replacement GPU card that provided the exact same functionality, did not affect the CPU's software, and was certifiable per DO-254. The FPGA GPU architecture described in this dissertation met all of those criteria and was selected as the path forward.

The basic requirement for this GPU circuit card is a VGA (640 x 480) resolution with 32-bit color (8 bits each for red, green, blue, and alpha) output. The host interface is a 33 MHz PCI bus, and the software graphics driver cannot change.

5.3.2 Design

The design of the FPGA-based GPU replacement card consisted of two separate efforts: circuit card design and FPGA design. The circuit card design was leveraged heavily from a recently designed I/O card that also used an FPGA to interface to the PCI bus. The FPGA design was based upon the previous GPU architectural work that was completed on an Altera Stratix III development kit. Each of these tasks is discussed in more detail below.

Circuit Card Design

The circuit card was designed in Cadence OrCAD schematic capture and Allegro PCB layout. The engineering staff at L-3 DS assisted in the detailed circuit card design, and an image of the PCB design is shown in Figure 44.

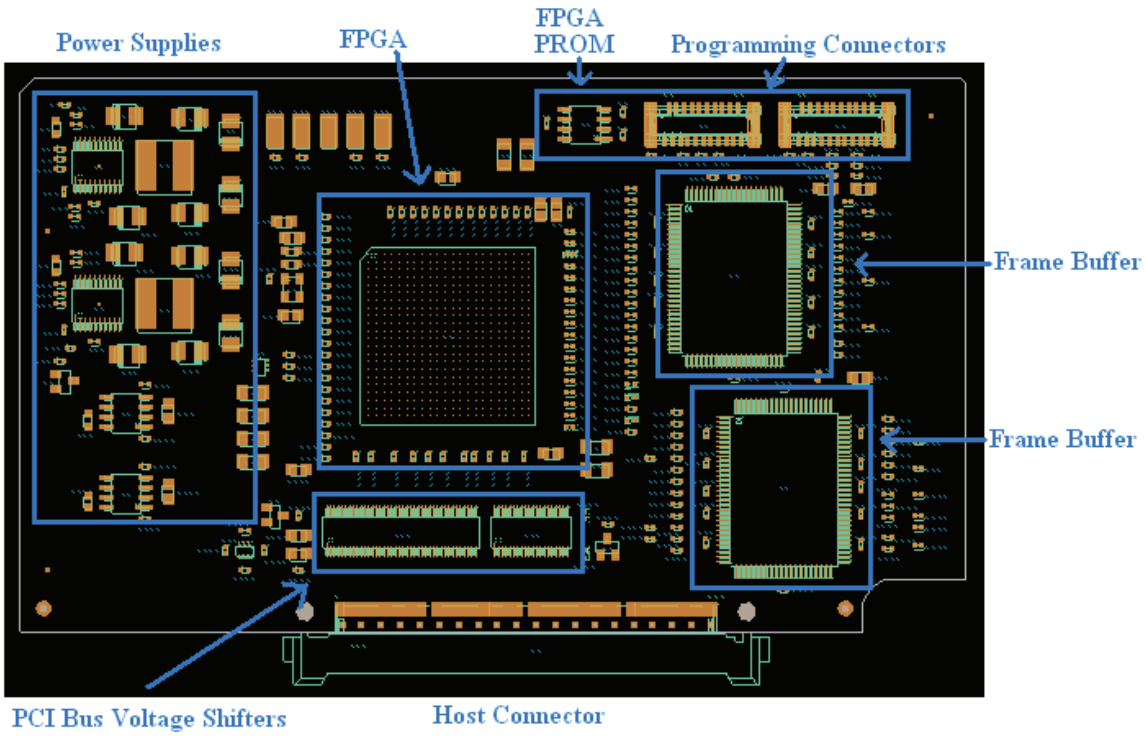


Figure 44: PCB layout of GPU circuit card.

The circuit card consists of the FPGA, configuration PROM, frame buffer memory, and power supplies. An Altera Cyclone III FPGA device (EP3C80F484I7) was chosen because of its use in a previous circuit card design. All of the power supply and configuration PROM circuitry was also leveraged from an existing design to accelerate the schedule. Because this circuit card will be used in an already established system, the 5V PCI bus interface had to be maintained. The Cyclone III FPGA is not 5V tolerant, so

high-speed voltage shifters were used to translate the 5V host interface to the 3.3V FPGA interface.

The mechanical packaging of avionics displays is usually very compact, and the small chassis does not typically allow easy debug and integration of new circuit cards. As such, an open-frame motherboard was designed that just provides the basic interconnect required to test the GPU functionality. The open-frame motherboard, shown in Figure 45, has a CPU connector and a GPU connector, with traces for the PCI interface between them. The motherboard also has a serial connector for the CPU interface and a DVI transmitter that converts the RGB output from the GPU into a DVI interface that can connect to a standard DVI monitor.

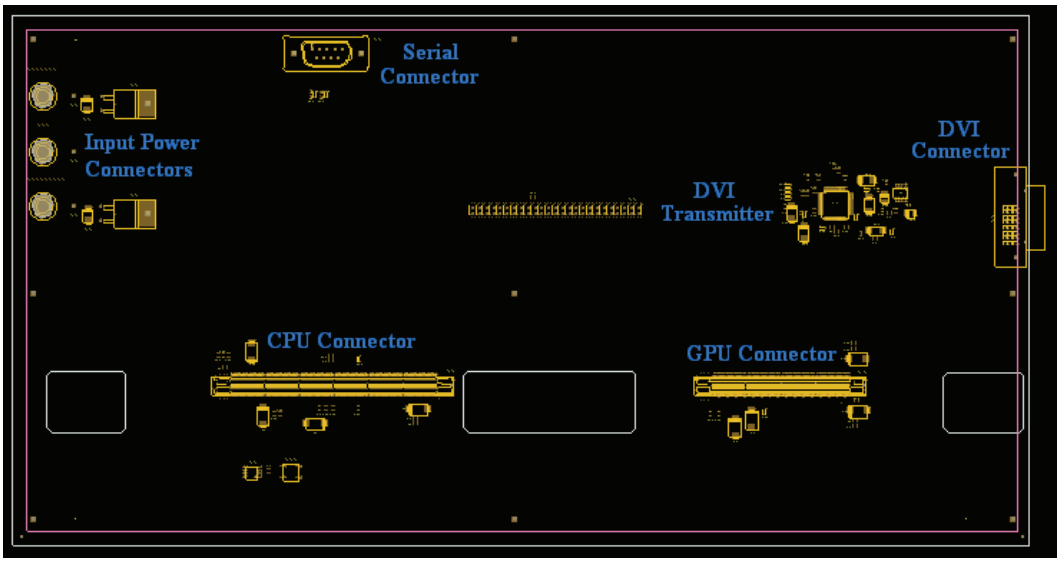


Figure 45: PCB layout of open-frame motherboard.

FPGA Design

The FPGA design was a collaborative effort with two other engineers at L-3 DS, and the FPGA GPU architecture described in Chapter 3 was customized in two ways to meet the specific requirements for this application. First, the host interface had to match the existing GPU's host interface exactly. This interface had to be maintained to avoid changing any of the CPU's software that has already been certified per avionics standards. Secondly, the memory interface changed from DDR2 to SRAM. This change took advantage of the lowered resolution (VGA) and greatly simplified the design. Had the resolution been SVGA or even XGA, then the original DDR2 memory interface would have been retained to meet the throughput requirements.

The block diagram of the FPGA GPU for this application is shown below in Figure 46. All of the blocks still perform the same functionality as described in Chapter 3. The PCI interface block was supplemented with the exact register set required to mimic the existing software interface. The Frame Buffer Manager was simplified because of the change to SRAM frame buffers instead of DDR2 SDRAM. SRAM has much lower latency than SDRAM, consumes less power, and also does not require periodic refreshes such as SDRAM. The disadvantage of using SRAM is that it is more expensive and typically has less capacity than DRAM. However, for this application, the simplified interface and reduced latency was more important than the cost or capacity.

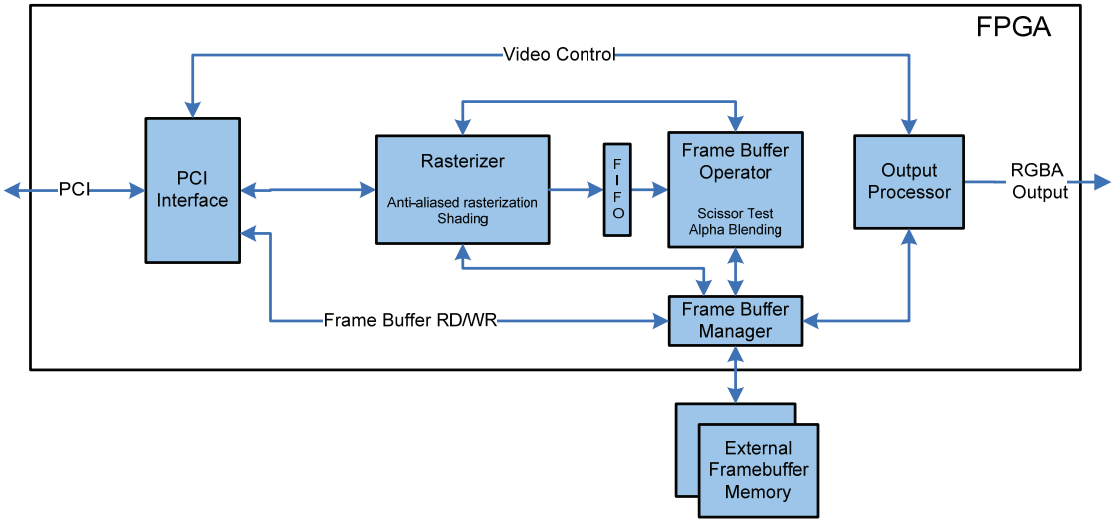


Figure 46: FPGA GPU block diagram.

As each module in the FPGA GPU architecture was modified for this specific application, it was simulated to validate its functionality. Finally when all of the modules were modified appropriately, the entire FPGA design was simulated from end to end. This simulation started with PCI bus transactions, tested the entire GPU pipeline, and used the previously described TXT2BMP utility to produce rastered images in Windows Paint. In order to stimulate the FPGA GPU with exactly the same interface provided by the CPU, the CPU's software was modified to dump the DMA buffers out a serial port. These DMA buffers were then captured and formatted for use by the FPGA simulation tool, ModelSim. Thus, the FPGA GPU design could be exercised with the exact same stimulus during simulation.

For example, Figure 47 shows the bitmap of one of the simulation outputs during the debug of the rasterization artifacts. The rasterized image shows horizontal line artifacts. After correcting some rounding errors in the Rasterizer and fixing the alpha

blending in the Frame Buffer Manager, the much improved image is shown in Figure 48. The usefulness of this end-to-end simulation with a graphical output cannot be emphasized enough. Many host interface bugs and rasterization artifacts were identified and resolved during the simulation – before the circuit card was even built.

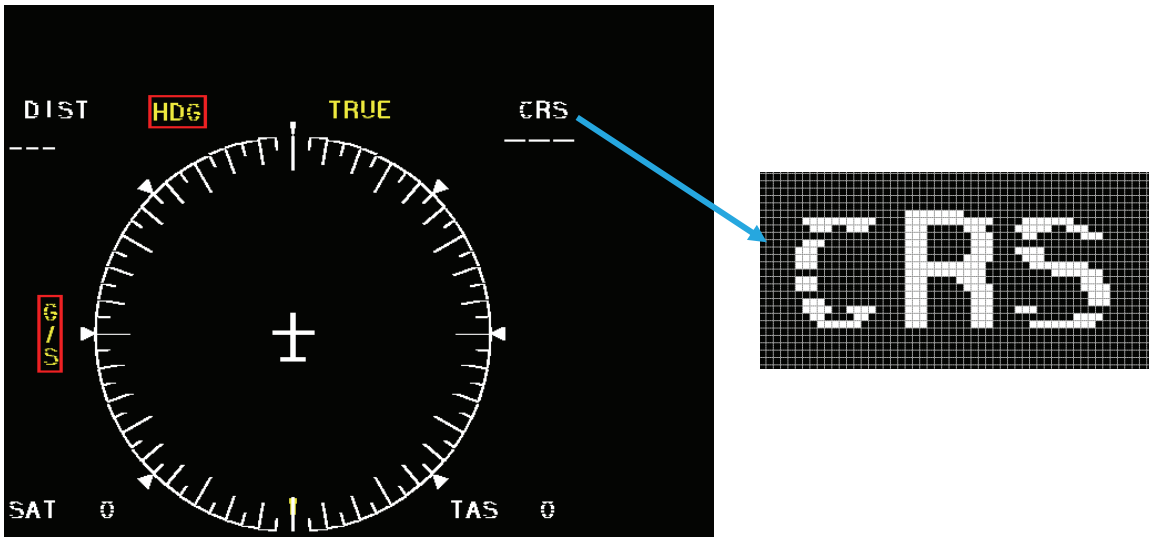


Figure 47: Simulation output of rasterized image with artifacts.

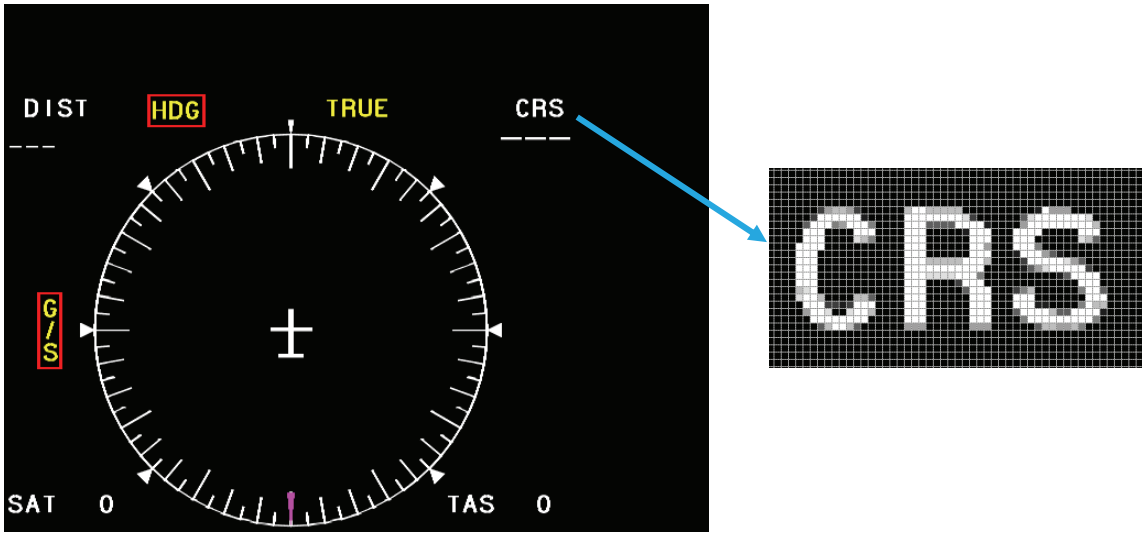


Figure 48: Simulation output of rasterized image with corrections.

During the FPGA modifications for this GPU application, the use of SRAM also motivated some further research into frame buffer management. Because this application's resolution was limited to VGA, each frame buffer device actually had enough memory capacity for two full displays. As previously discussed in Chapter 3, the delay of clearing the frame buffer can consume a substantial amount of frame time. To optimize this operation, four frame buffers can be used and the buffer clearing can be accomplished in the background, in an otherwise unused buffer. Figure 49 depicts that buffers 1 and 2 are physically located in the first memory device, and buffers 3 and 4 are physically located in the second memory device. During each frame, only three of the buffers are used – one for clearing, one for filling (rendering), and one for the display output (reading).

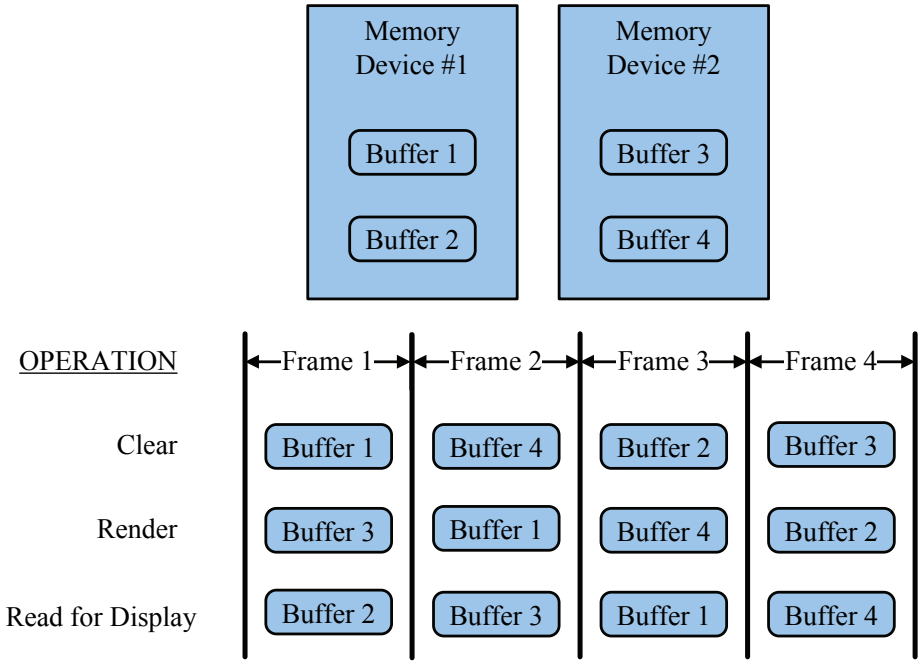


Figure 49: Frame buffer management with four buffers.

The sequence of buffers is set up such that the clearing and reading are always in the same physical memory device. This arrangement was made to maximize the memory bandwidth to the device that is used for rendering. The Frame Buffer Manager was modified to actually perform the clearing concurrently with the reading. As the pixel location increments through each frame, the corresponding pixel data is read from one buffer in the memory device and cleared in the other buffer. In the end, the memory bandwidth to the GPU pipeline is completely unaffected by the reading and clearing operations because the other memory device is used for the reading and clearing. All of these optimizations were possible because of the switch to the SRAM devices with low latencies.

5.3.3 Implementation

The final GPU circuit card is shown in Figures 50 and 51 as installed in the open-frame motherboard, next to the CPU card. The small green circuit card is the programming adapter board for the FPGA GPU. The prototype cards are fabricated with a red solder mask to ensure that they are not mistakenly used in a flight-worthy product. The green CPU card shown next to the GPU in Figure 51 is a flight-worthy card designed by the author in 2002.

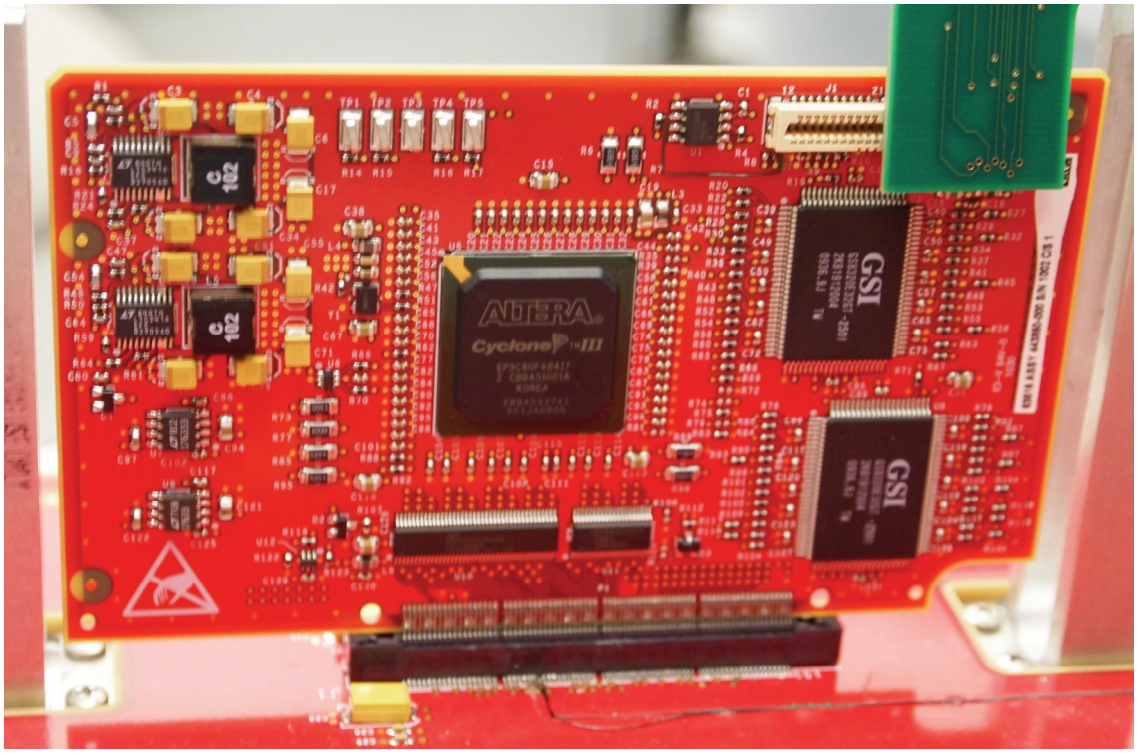


Figure 50: Picture of FPGA GPU circuit card.

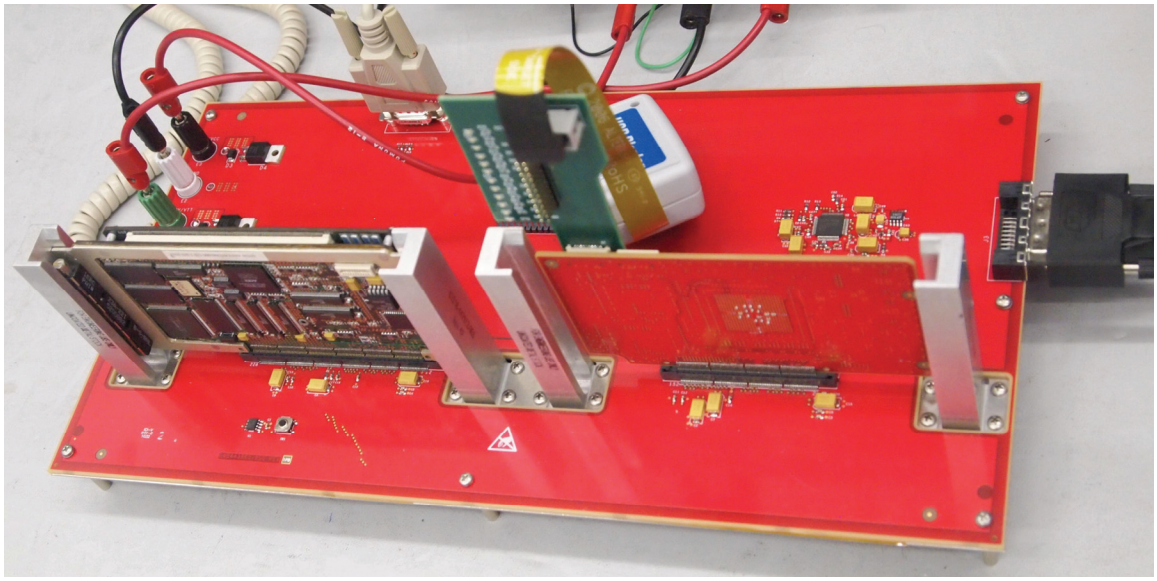


Figure 51: Picture of GPU and CPU circuit cards installed in motherboard.

The integration of the FPGA-based GPU into the existing software and infrastructure was begun by first establishing PCI communication with the FPGA. Next a series of memory tests were developed in the CPU's software to exercise both the frame buffer memory and the GPU registers. After a few changes to the settings of the PCI core in the GPU, the PCI interface has been extremely consistent and dependable. Next, some simple graphics commands were utilized to clear the screen with a specific color and then to swap the frame buffers. Finally, some demo display pages were generated to exercise the Rasterizer and full GPU pipeline. Due to the previous end-to-end simulation of the GPU with the exact same PCI transactions, there were very few rasterization artifacts or anomalies to correct in the FPGA.

The functionality was successfully integrated together with the CPU software to produce the demonstration display as shown in Figure 52. The demonstration software emulates a horizontal situation indicator (HSI) and rotates the graphical symbology to simulate a turning aircraft. The update rate is 60 Hz, which is consistent with the refresh requirements for other aircraft displays.

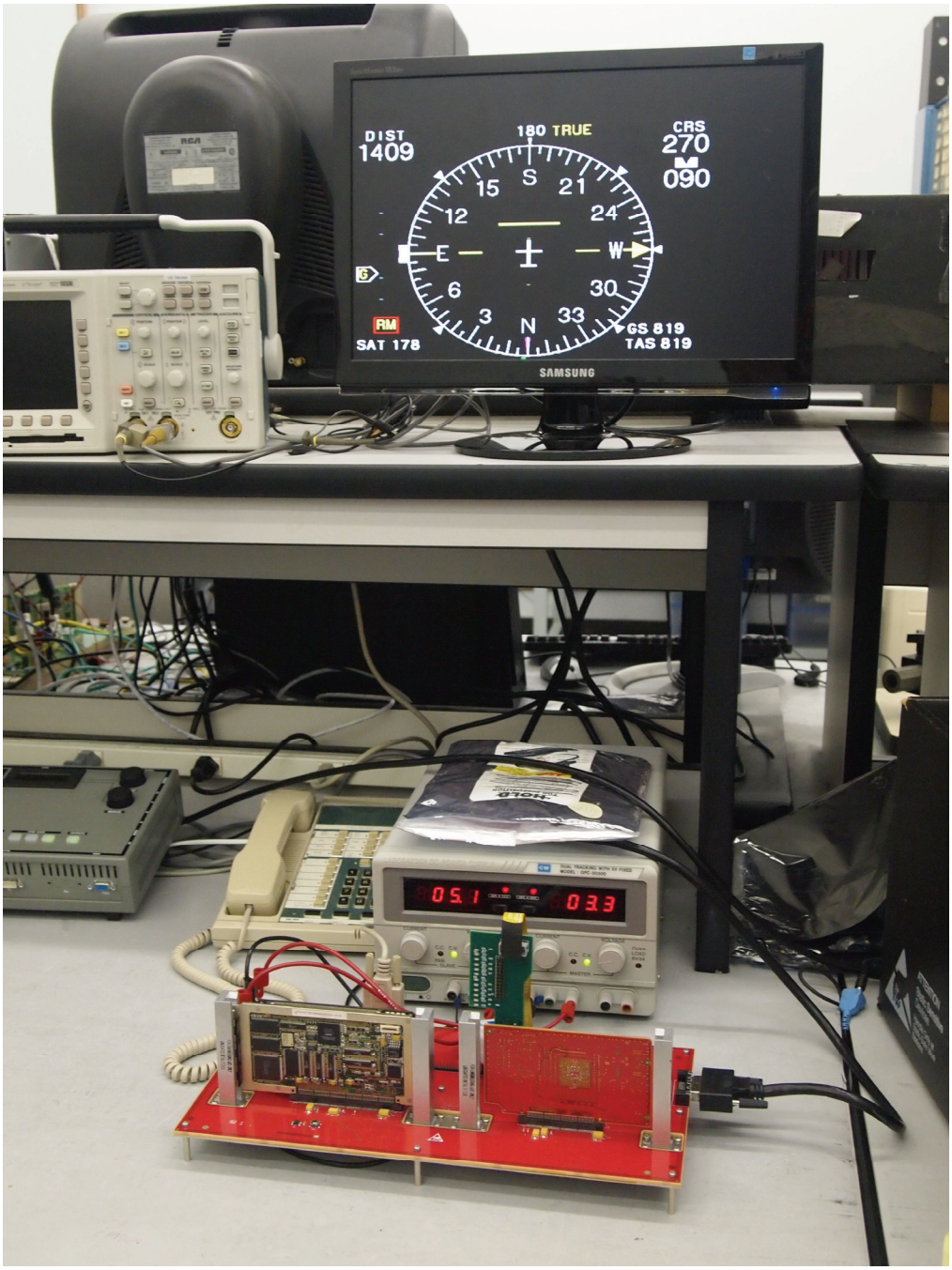


Figure 52: Picture of lab setup with demonstration display.

The final FPGA design uses less than 20% of the available logic elements in the EP3C80 device. The design uses an external 40 MHz clock and an internal PLL to run at a speed of 100 MHz, which was throttled down to take advantage of reduced power. A summary of the FPGA's resource utilization is shown in Table 8.

Table 8. Resource utilization summary.

Module	Logic Cells	Memory Bits	DSP Blocks
PCI Interface	2128	149280	
Rasterizer	11326		64
Frame Buffer Manager	515	1368	
Frame Buffer Operator	221	1696	2
Output Processor	154		
TOTAL	**14344**	**152344**	**68**

Measurements were taken with an oscilloscope to confirm the simulation results that indicated the FPGA GPU can render the avionics display symbology in 2.2 milliseconds. When the display uses a typical frame rate of 16.67 msec/frame, this leaves nearly 14.5 milliseconds or more than 87% of the frame time available for additional graphical symbology. In comparison, the original GPU required 6.5 milliseconds for rendering the same display and thus had only 61% available throughput remaining. Thus, the FPGA GPU provided a 3x speed-up in graphics performance over the original GPU ASIC.

5.3.4 Comparison with Conventional Methods

The FPGA GPU implementation was compared with existing GPU designs to validate the FPGA-based approach. The 3DLabs GLINT GPU was discontinued in 1999, but it provides basic graphics functionality that still satisfies many avionics applications. The AMD M9 GPU, which has also been discontinued because it was primarily developed for laptops, provides a higher level of functionality and is used routinely in avionics because ALT software has a certified OpenGL driver for its interface. The results of the comparisons are shown in Table 9 and Figure 53.

Table 9. Comparison of GPU approaches.

	FPGA	GLINT	M9
Total Power (W)	**5**	**12**	**18.1**
GPU	2.2	3.75	14.7
Frame Buffers	1.75	6	0
Power Supplies	1.05	2.25	3.4
Total Cost ($)	**605**	**912**	**390**
GPU	335	422	165
GPU Accelerator	0	198	0
Frame Buffers	105	112	0
FPGA PROM	15	0	0
Power Supplies, PWB, Heatsinks, etc	150	180	225
Total Board Area (sq inches)	**12**	**17.4**	**15.6**
Total Board Weight (g)	**79**	**111**	**158**

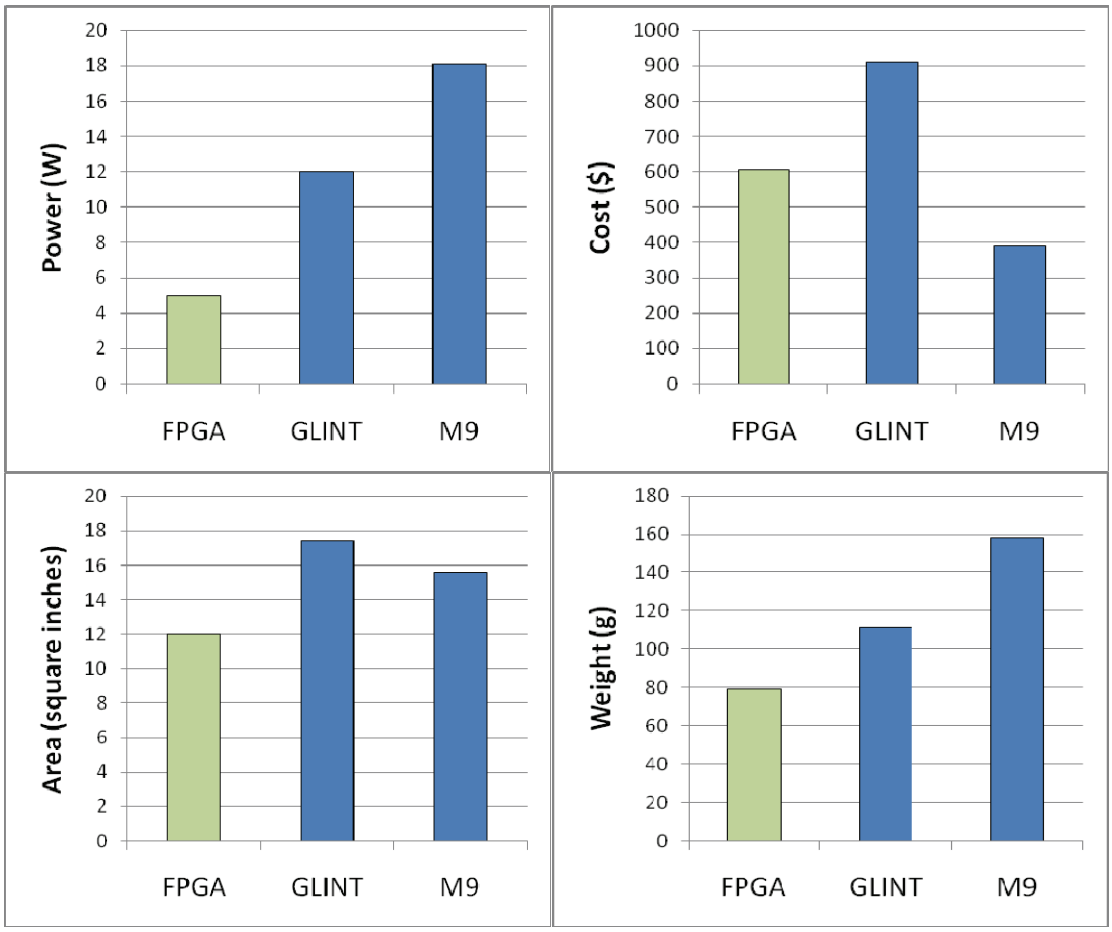


Figure 53: Comparison of GPU implementations.

The FPGA-based solution offers considerable power, area, and weight reductions over the other two implementations. The FPGA GPU reduces the power consumption by 60% over the GLINT and nearly 75% over the M9. The FPGA requires less power than the GLINT or M9 solutions because the FPGA has been designed exactly to meet the requirements of the avionics application. The FPGA is not burdened with additional functionality to accommodate other applications. The increased power consumption of the GLINT and M9 GPUs require over-the-top heat sinks to locally pull the heat off the chips and towards the chassis wall. The FPGA GPU solution, on the other hand, does not

require any additional heat sinking. This advantage contributes to a weight reduction of 50% from the M9 and a cost reduction of nearly 45% from the GLINT implementation.

The FPGA GPU solution also requires less circuit card area than the other two GPUs. The GLINT GPU requires eight VRAM devices for the frame buffer whereas the FPGA GPU only need two SRAM devices. The M9 GPU has embedded DRAM for the buffers but still requires more overall area because of the package size and additional power supply needs. The circuit card area savings is 25-35% when choosing the FPGA GPU over the other options.

The cost data, which was collected for quantity 100 pricing to model a typical avionics application, shows that the M9 GPU implementation costs less than the FPGA GPU. The M9 solution is less expensive mainly because of the embedded DRAM and high volume production of the M9 GPU. However, because the M9 GPU is obsolete and supplies are limited, its cost is expected to rise in the coming years.

5.3.5 Summary

The FPGA-based GPU architecture presented in Chapter 3 has been successfully applied to an avionics display. The rasterization performance supplies significant throughput margin for additional graphical symbology and provides a 3x speedup over the existing GLINT GPU. The FPGA GPU's power consumption is also decreased by 60% and the cost decreased by 45% relative to the GLINT GPU. In addition, the FPGA device availability is much improved over the ASIC GPU solutions, which leads to a less expensive life cycle cost for the product.

The FPGA device chosen for this application also has nearly 80% of the logic resources remaining for further functionality such as soft-core CPUs or specific avionics interfaces. The overall success of this FPGA GPU provides an excellent platform for future expansion of graphics, video, and interfaces that can be leveraged in other avionics applications. Alternatively, the FPGA GPU could be hosted in a smaller FPGA to reduce costs and power even further.

CHAPTER 6

FPGA GPU EXTENSIONS FOR COMPREHENSIVE GRAPHICS

The basic FPGA GPU architecture presented in Chapter 3 provides all of the functionality required for many industrial, embedded, or portable applications. To demonstrate its scalability to broader applications, the architecture can be extended to accommodate the vertex processing, fragment processing, and even streaming video as textures. The result is a GPU pipeline described in this chapter that provides hardware acceleration for all of the functionality in the OpenGL Safety Critical (SC) specification. The hardware acceleration is significant because many GPUs actually depend on their software drivers to provide the vertex processing or fragment processing. These software implementations constrain performance because they do not take full advantage of the parallelism available in a GPU. Through the extensions shown in this chapter, all of the OpenGL functionality is provided with true hardware acceleration in the FPGA.

The initial approach for the full OpenGL SC GPU pipeline resulted in an architecture with multiple soft-core processors, as shown in Figure 54. The Nios soft-core processor controls the sequencing and operation of the pipeline, and the smaller multi-threaded processors (shown in orange) handle the specific tasks along the pipeline. The mathematically intensive portions of each specific task could be offloaded to the hardware acceleration blocks (HABs). However, this architecture was still heavily dependent upon the performance of the Nios as it manages data flow through the

pipeline. In fact, this approach somewhat mimicked the GPUs that depend upon their software drivers for key functionality. The desire to have more hardware accelerated processing led to the decentralized approach based upon the basic GPU architecture presented in Chapter 3.

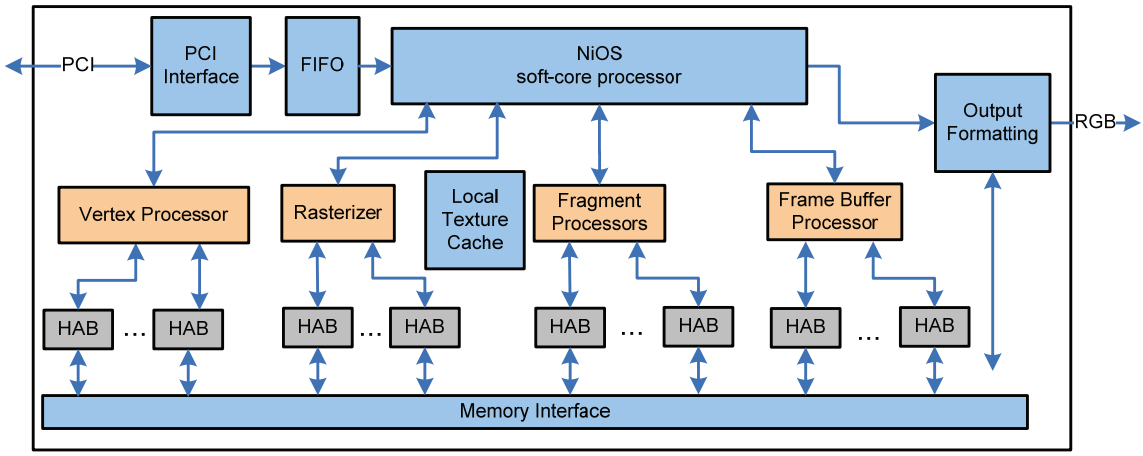


Figure 54: Initial approach for comprehensive graphics processing.

The recommended architectural approach for a full OpenGL SC GPU pipeline is shown in Figure 55, with the blue blocks indicating functionality that has been reused from the basic GPU architecture. This architecture uses the same FIFO-based approach to interconnect the modules in the pipeline. The new pipeline modules for the comprehensive graphics functionality are the Vertex Processor and the Fragment Processor. To support the increased functionality, a Texture Manager and a Video Controller have also been added. Each of these four new blocks is defined in more detail through the remainder of this chapter.

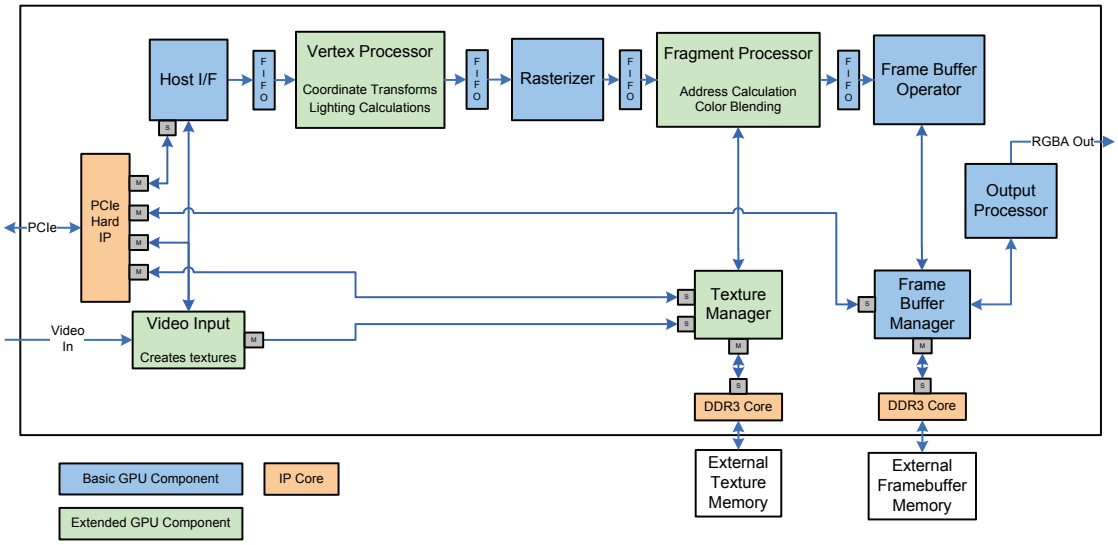


Figure 55: Extended architecture for comprehensive graphics processing.

6.1 Vertex Processor

The Vertex Processor is responsible for transforming vertex coordinates and applying lighting calculations to vertices. As each vertex enters the Vertex Processor, it consists of a three-dimensional coordinate and a base color (red, green, blue, and alpha). Based upon the settings of the Vertex Processor, the coordinates and the color will be transformed before the vertex outputs from this module. The transforms and lighting algorithms are well defined by OpenGL, but their implementations are not specified. A soft-core processor could be used to handle the vertex processing with maximum flexibility, but the significant mathematical burden would be best suited for hardware acceleration. The remainder of this section is segregated into portions that first discuss the coordinate transformation and then deal with the lighting calculations.

6.1.1 Coordinate Transformations

The coordinate transformations are based upon two user-defined 4x4 matrices, the modelview matrix and the perspective matrix. Initially, the vertices are received from the host application with their coordinates in modeling space. These vertex coordinates are first multiplied by the modelview matrix to transform the coordinates into eye space. Next the coordinates are multiplied by the perspective matrix to obtain the vertex in perspective coordinates. The perspective factor (w') that results from this perspective multiplication ([x y z 1] -> [x' y' z' w']) is used to divide into each of the XYZ coordinates to provide the perspective correction. The final coordinate transformation scales the perspective coordinates (range from [0,0] to [1,1]) into the defined viewport resolution. The result is a set of coordinates that are defined in the range of actual screen coordinates and therefore can be used by the Rasterizer.

The architecture for implementing the vertex processing could be optimized for speed or area. For optimum speed, many of the calculations need to be completed in parallel at the expense of an increase in the required FPGA resources. However, since vertices are processed much less often than fragments (there are typically many fragments generated per set of vertices), the vertex processing architecture was optimized based on area, or FPGA resources. The block diagram of the coordinate transformation calculations is shown in Figure 56.

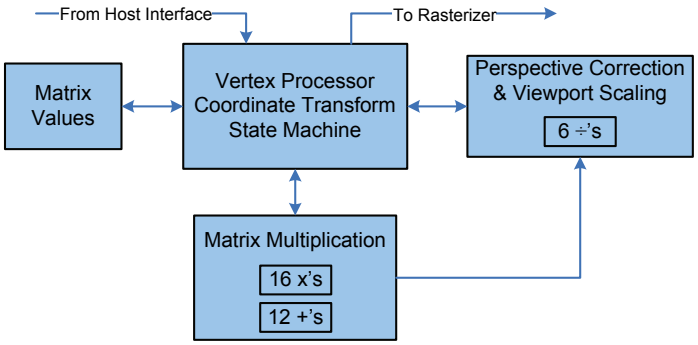


Figure 56: Architecture for vertex coordinate transformation.

Each matrix multiplication requires sixteen multiplications and twelve additions, which together consumes eight DSP blocks in the Altera Stratix IV. The architecture shown above uses the Matrix Multiplication module to perform both the modelview and perspective transformations. The Coordinate Transform State Machine stores the matrix values from the host interface and then multiplexes them into the Matrix Multiplication based upon the stage of vertex processing. This resource reuse means that only one vertex can be processed at a time. However, the six-cycle latency to complete each coordinate transformation is relatively short compared to the amount of time spent rasterizing and processing fragments based upon the vertices.

6.1.2 Lighting Calculations

The OpenGL SC lighting equation is shown below in Equation 6.1, and Table 10 defines the variables. The emissive and ambient contributions are listed at the beginning of the equation, and then the contribution from each light in the scene is summed together. The mathematical calculations required for diffuse and specular lights are

significantly more involved than the ambient lights. Diffuse lights require the dot product of a normal vector with the light position vector. Specular lights require the dot product of the normal vector with a half vector, which is the summation of the light and eye vectors. In addition, specular lights require an exponential multiplication which is limited to an integer from 0 to 128 in this architecture. Each of the variables is passed into the Vertex Processor by the Host Interface. The summation limits the scene to eight light sources, but this maximum is arbitrary based upon knowledge of requirements from reasonable applications. A similar approach could support more than eight light sources at the expense of additional delay based upon the number of lights.

$$\text{Vertex color} = M_E + (S_A \times M_A) + \sum_{8 \text{ lights}} [(L_A \times M_A) + (L_D \times M_D \times N \bullet L) + (L_S \times M_S \times (N \bullet H)^{Sh})] \quad (6.1)$$

Table 10. Lighting equation variables.

Variable	Description
M_E	material emissive color
S_A	scene ambient color
M_A	material ambient color
L_A	light ambient color
L_D	light diffuse color
M_D	material diffuse color
L_S	light specular color
M_S	material specular color
N	normal vector in eye space (normal transformed by inverse modelview matrix)
L	light position in eye space (coordinates transformed by modelview matrix)
H	half-vector between the light and eye positions (light vector + eye vector)
Sh	material shininess factor (restricted to integer 0-128)

The lighting calculation requires several operations that are not standard combinatorial or sequential logic in an FPGA. The normal vectors, dot products, cross products, and exponential multiplication all required analysis to devise an approach in the FPGA architecture. The normal vector is essentially a cross product between the two vectors associated with the vertex. The cross product has been decomposed into the simplification shown in Equation 6.2 because the vectors are known to be three dimensional.

$$\begin{aligned} N = V_1V_2 \times V_1V_3 = ([V_{2y}-V_{1y}]*[V_{3z}-V_{1z}] - [V_{2z}-V_{1z}]*[V_{3y}-V_{1y}], \\ [V_{2z}-V_{1z}]*[V_{3x}-V_{1x}] - [V_{2x}-V_{1x}]*[V_{3z}-V_{1z}], \\ [V_{2x}-V_{1x}]*[V_{3y}-V_{1y}] - [V_{2y}-V_{1y}]*[V_{3x}-V_{1x}]) \end{aligned} \quad (6.2)$$

In the lighting equation, the normal vector is always used in conjunction with a dot product, so the further simplification was made as shown in Equation 6.3

$$\begin{aligned} N \bullet M = N_1M_1 + N_2M_2 + N_3M_3 = ([V_{2y}-V_{1y}]*[V_{3z}-V_{1z}] - [V_{2z}-V_{1z}]*[V_{3y}-V_{1y}])*M_1 \\ + ([V_{2z}-V_{1z}]*[V_{3x}-V_{1x}] - [V_{2x}-V_{1x}]*[V_{3z}-V_{1z}])*M_2 \\ + ([V_{2x}-V_{1x}]*[V_{3y}-V_{1y}] - [V_{2y}-V_{1y}]*[V_{3x}-V_{1x}])*M_3 \end{aligned} \quad (6.3)$$

In addition to the normal vector calculations and dot products required, the vector processing also requires matrix inversion and square root operations. Other FPGA researchers have already developed efficient algorithms for handling these calculations, and their work was leveraged for this architecture [94, 95].

Once all of the approaches were devised for accomplishing the required mathematical operations, the overall calculations had to be sequenced and controlled to produce the final result. The state machine shown in Figure 57 essentially acts as a large while loop to sum together the contribution from each of the eight lights. Based on the latency for each calculation in the equation, the state machine latches the intermediate outputs as soon as they are available. For example, when the specular light's shininess factor is greater than one, multiple clock cycles are required to iteratively perform the exponential operation as a sequence of multiplications.

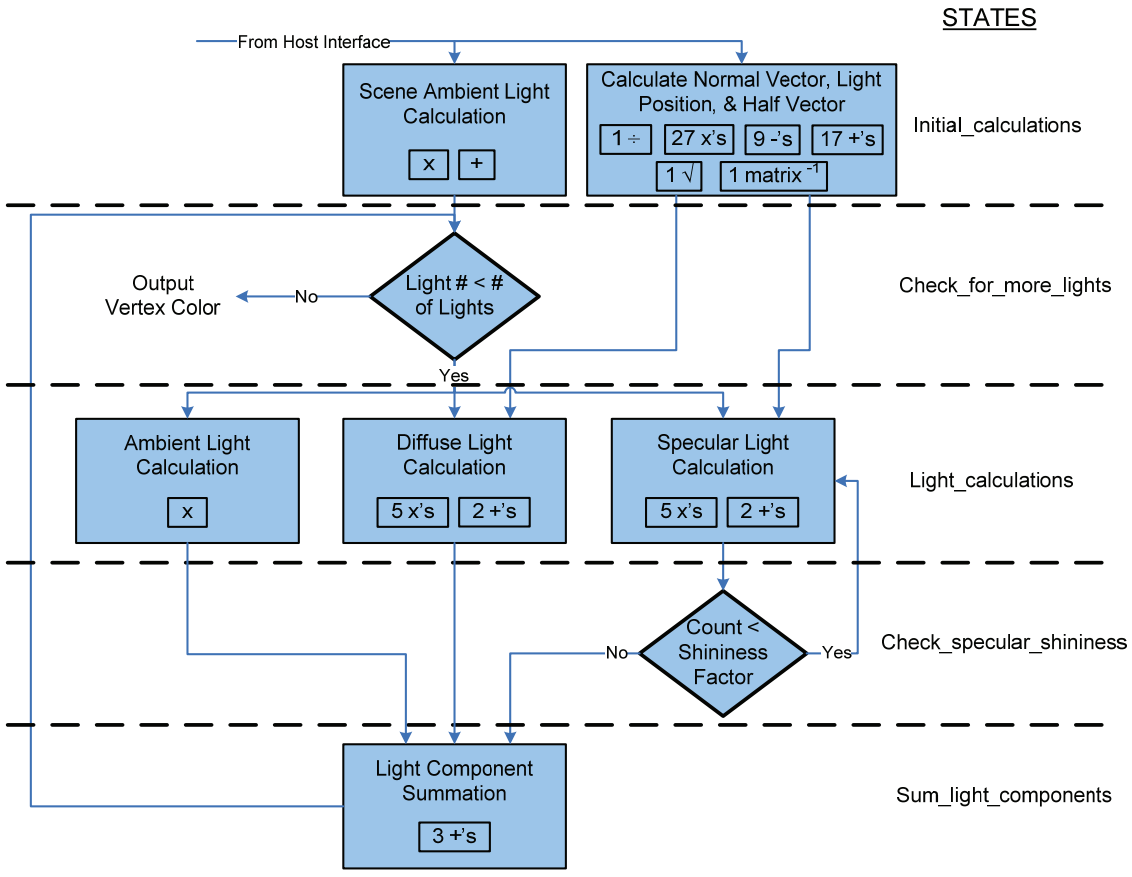


Figure 57: Architecture for vertex lighting calculations.

6.2 Fragment Processor

The Fragment Processor is responsible for manipulating the color data of each pixel in accordance with various texture mode commands. The pixel data and commands are both received from the Rasterizer, and the output pixel data is sent on to the Frame Buffer Operator. The most often used functionality in the Fragment Processor is that of texture blending. A texture is a set of data in memory that can be thought of as a bitmap, although it is not always in this exact RGB format. The Fragment Processor blends the color data from the texture with the pixel data to create a new pixel to output to the Frame Buffer Operator.

Because the Fragment Processor would benefit from a certain level of programmability that would allow shaders or other custom functionality, the initial approach for the Fragment Processor used a soft-core processor. However, analysis quickly revealed that millions of fragments per second could not be effectively processed by a soft-core processor executing at approximately 150 MHz. As such, a new hardware-based approach was developed for the Fragment Processor, and its architecture is shown in Figure 58. The detailed design of the Fragment Processor was completed by another researcher at L-3 Communications.

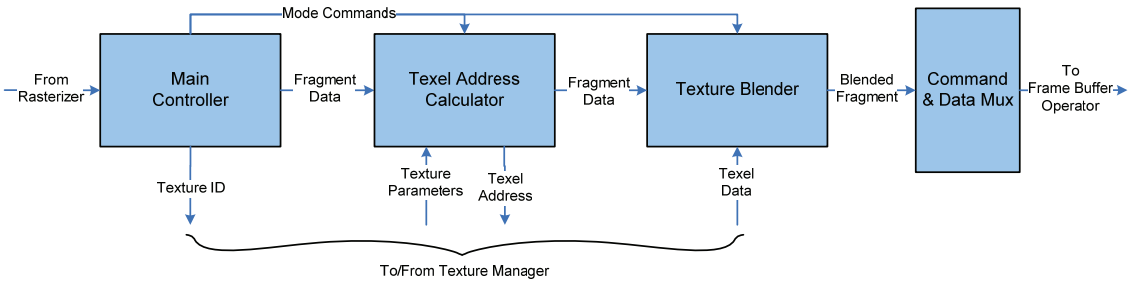


Figure 58: Fragment Processor block diagram.

The Fragment Processor consists of three major functional modules with a multiplexer at the end to handle the output to the Frame Buffer Operator. The primary modules are the Main Controller, Texel Address Calculator, and the Texture Blender. The Main Controller provides overall control for the Fragment Processor. The Texel Address Calculator generates the texel (texture element) addresses and sends them to the Texture Manager. The Texture Blender actually blends the fragment color data with the texel color data to produce the final fragment output. Each of these modules is described in more detail in the following sections.

6.2.1 Main Controller

The Main Controller (MC) is responsible for controlling the overall operation of the Fragment Processor. The MC receives OpenGL pipeline commands from the Rasterizer (via a FIFO) and maintains all OpenGL state variables. The MC uses a large state machine to synchronize the processing and to control the flow of data to and from the Fragment Processor. The list of commands supported by the Fragment Processor is shown in the table below.

Table 11. Fragment Processor commands.

Command	Data Word 1	Data Word 2	Data Word 3
Enable/Disable	2D Texture		
Bind Texture	<Texture ID>		
Texture Environment	Mode	Modulate, Replace, Add, Decal, or Blend	
	Color	<RGBA>	
Texture Parameter	Mag Filter	Nearest or Linear	
	Texture Wrap – S	Repeat or Edge Clamp	
	Texture Wrap – T	Repeat or Edge Clamp	
Fragment	<X & Y & Z>	<RGBA>	<S & T>

The only command that initiates any operation in the Fragment Processor is the fragment command. All other commands are only used to update the state variables that are stored by the MC. When the fragment is received, the current state of all variables is used by the actual fragment operations. If the texture is in the disable state when a fragment is received, then the fragment data is passed on to the Frame Buffer Operator without any processing. When texture environment or parameter changes are received, the MC will stall any new fragments until the Fragment Processor has output all fragments that are currently in process. This method of stalling the processing during a state change ensures that the strict ordering of data and commands is maintained as they fragments flow to the Frame Buffer Operator.

6.2.2 Texel Address Calculator

The Texel Address Calculator (TAC) determines the addresses of the texels required for blending operations. For each fragment, there can be either one or four texels required, depending on the magnification filter value. If the nearest magnification mode is selected, then only one texel is required for blending; however, if the magnification is set to the linear mode, four texels are required along with interpolation. The texel addresses are offset values from the base of the selected texture's memory address. The equations used for the texel offset calculations are shown in Equations 6.4 and 6.5, assuming that S and T are in the range from 0 to 1.

Nearest Magnification:

$$\text{offset} = [\text{round}(S*\text{texture\_width})] + [\text{round}(T*\text{texture\_height})]*\text{texture\_width} \quad (6.4)$$

Linear Magnification:

$$\text{offset1} = [\text{roundup}(S*\text{texture\_width})] + [\text{roundup}(T*\text{texture\_height})]*\text{texture\_width}$$

$$\text{offset2} = [\text{roundup}(S*\text{texture\_width})] + [\text{rounddown}(T*\text{texture\_height})]*\text{texture\_width}$$

$$\text{offset3} = [\text{rounddown}(S*\text{texture\_width})] + [\text{roundup}(T*\text{texture\_height})]*\text{texture\_width}$$

$$\text{offset4} = [\text{rounddown}(S*\text{texture\_width})] + [\text{rounddown}(T*\text{texture\_height})]*\text{texture\_width} \quad (6.5)$$

If S and T are not in the range of [0, 1], then they must be shifted to that range according to the texture wrap parameters. If repeat mode is selected, then any S value outside of [0, 1] is shifted into that range by subtracting or adding a multiple of the texture width. A similar algorithm is followed for the T value by shifting with the texture height. If clamp mode is selected, then the actual S and T values are clamped to the [0, 1] range.

To implement the calculations required, a six-stage FPGA pipeline was developed. This architectural technique ensures that each calculation will be complete in six clock cycles but also allows multiple (up to six) calculations to be underway simultaneously. The pipeline diagram for the texel offset calculations in shown in Figure 59.

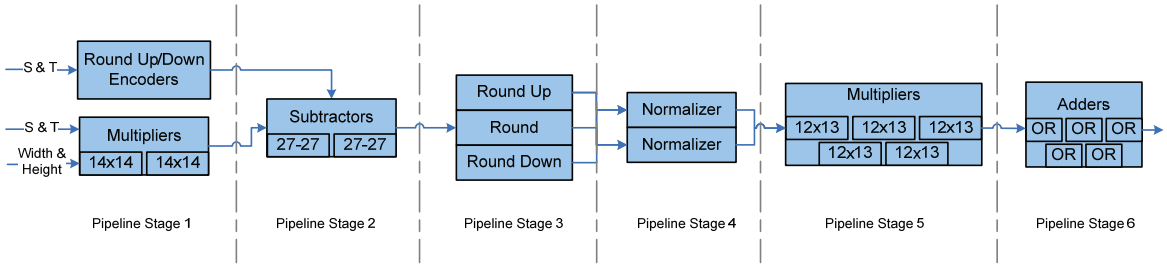


Figure 59: Texel address calculation pipeline diagram.

In the first stage, the S and T values are multiplied by their respective dimensional limits (width or height) at the same time that an encoder determines the adjustment that is necessary when S or T is out of the [0, 1] range. The multipliers use the FPGA's DSP blocks to perform 14-bit multiplication without consuming logic elements. In the second stage, simple 27-bit subtraction translates the texel back into the width and height of the texture. In the third stage, parallel processes perform three different types of rounding. In the fourth stage, normalization prepares the data for the multiplication, which occurs in the fifth pipeline stage. This stage also uses the FPGA DSP blocks to perform 12-bit by 13-bit multiplication. The final stage of the pipeline performs a logical OR operation to add the multiplier's output with the normalized x-value of the offset.

6.2.3 Texture Blender

The Texture Blender (TB) performs the color blending based upon the fragment data and texel data. To accommodate the texture linear magnification mode, there are two steps in the color blending process as shown in Figure 60. The nearest magnification mode bypasses the first step and proceeds directly with the second step of actual color blending.

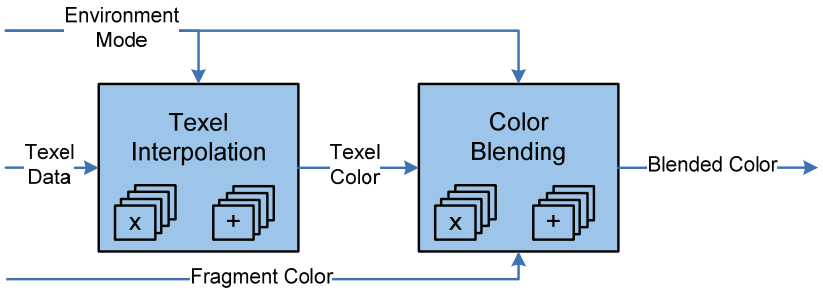


Figure 60: Texture blending diagram.

The first step blends the four texel colors together using a weighted average calculation to determine the interpolated texel color. This module uses a combination of seven multipliers and adders for each color component, with all four components being processed simultaneously to maximize throughput. The second step blends together the texel color and fragment color based upon the texture environment mode setting. Each setting requires a different equation as shown in Table 12, and each color component requires four multipliers and five adders.

Table 12. Texture blending modes.

Mode	RGB Blending	Alpha Blending
Modulate	$C_O = C_T * C_F$	$A_O = A_T * A_F$
Replace	$C_O = C_T$	$A_O = A_T$
Add	$C_O = C_F + C_T$	$A_O = A_F * A_T$
Decal	$C_O = C_F * (1 – A_T) + C_T * A_T$	$A_O = A_F$
Blend	$C_O = CF * (1 – C_T) + C_C * C_T$	$A_O = A_F * A_T$
C_O = color output, C_F = fragment color, C_T = texture color, A_O = alpha output, A_F = fragment alpha, A_T = texture alpha, C_C = environment color		

To optimize the timing performance, the blended color output for each environment mode is calculated all the time, and multiplexers determine which blended color to output from the Fragment Processor. Overall, the Texture Blender requires 44 multipliers and 48 adders to complete the texel interpolation and fragment blending. While the texture calculations are ongoing, the unneeded fragment data (XYZ position) is stored in a FIFO that is positioned in parallel with the processing. When the final calculation is ready, the FIFO's data is read and combined with the calculated color for output from the Fragment Processor.

6.3 Texture Manager

The Texture Manager coordinates the interfaces between the texture memory, the Fragment Processor, the Video Controller, and the PCI Express port. Its primary focus is reducing the amount of time that the Fragment Processor is stalled while waiting for texture data from the external texture memory. The Video Controller and the PCI Express port both provide data into the texture memory, and the Fragment Processor is typically the only interface reading data out of the texture memories. Texture data is available for reading by the PCI Express port as well, but this interface is not as impacted by latencies. The architectural approach for the Texture Manager is shown in Figure 61, and its detailed design was completed by another researcher at L-3 Communications.

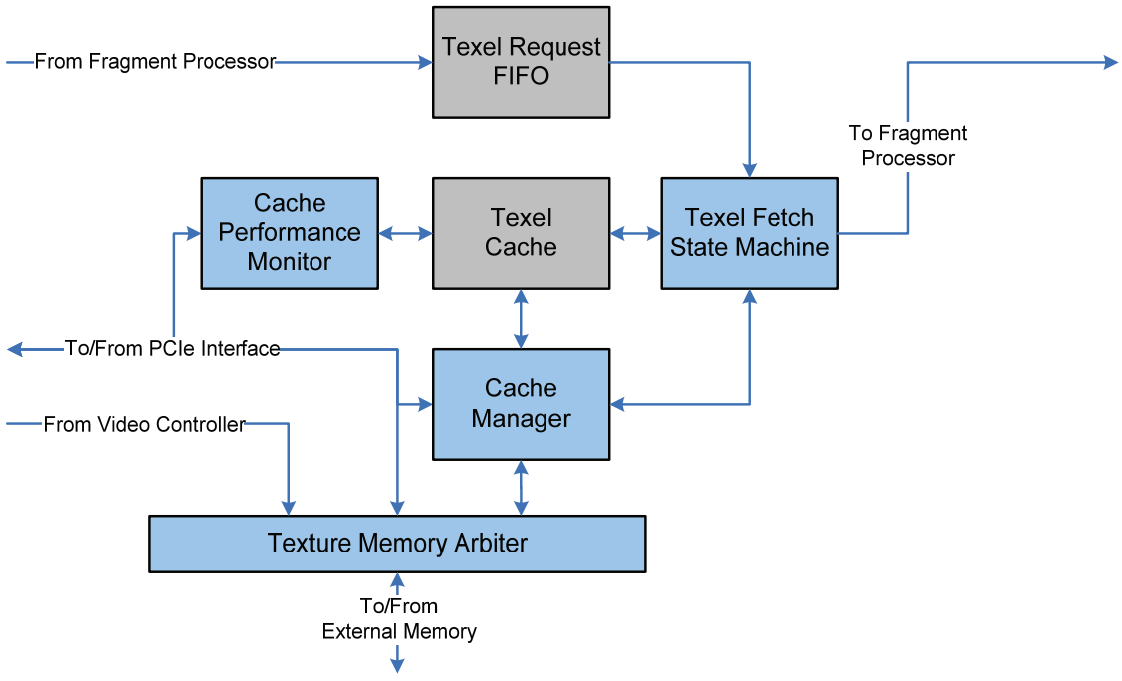


Figure 61: Texture Manager block diagram.

The Texture Manager consists of four major modules shown in blue in addition to the two storage elements shown in gray. The Texel Fetch State Machine reads texel requests from the FIFO and retrieves the appropriate texel data from either the Texel Cache (if it is available there) or the Cache Manager. The Cache Manager coordinates texel data reads from texture memory but also continuously fills the texel cache during idle times on the texture memory interface. The Texture Memory Arbiter has three Avalon-MM interfaces as inputs and one Avalon-MM interface as an output to the external texture memory. The Cache Performance Monitor continuously reads texel cache validities and maintains statistics on cache hits and misses. Each of these major modules is described in more detail in the following sections.

6.3.1 Texel Fetch State Machine

The texel requests from the Fragment Processor are first stored in the Texel Request FIFO and then read out by the Texel Fetch State Machine. Based upon the texel address, the state machine first looks to see if the Texel Cache has valid data for that address. If so, the state machine reads the Texel Cache and quickly sends the data back to the Fragment Processor. If the data is not available in the Texel Cache, then the state machine sends the texel address to the Cache Manager for data retrieval. The state machine uses a very simple design that can provide texel data to the Fragment Processor within three clock cycles if the data is available in the cache. If the data is not in the cache, then the latency is dependent upon the available bandwidth and latency of the texture memory interface.

The Texel Fetch State Machine was optimized by pipelining its operation into two stages. The first stage checks the Texel Cache for the available data, and the second stage stalls while waiting on the data to return after the Cache Manager has retrieved the data from external memory. The primary advantage to this approach is that if one texel request has stalled while awaiting data from external memory, the first pipeline stage can continue processing new texel requests as long as the data is available in the cache.

6.3.2 Cache Manager

The Cache Manager is responsible for maintaining the most useful texel data in the cache as well as for fielding new data requests from the Texel Fetch State Machine. The flexibility of an FPGA-based approach allows the use of different cache management techniques based upon the application's requirements. For applications that rely upon small textures to provide details in many different objects, the cache is best managed as several small sub-caches. Each sub-cache would contain part or all of the data from each of the small textures. For applications that typically apply large textures to surfaces on the display, the cache is best managed as one large cache that contains the data from multiple rows of the same texture. Keeping the texture's row data together is important because the Rasterizer outputs horizontal lines during its triangle fill operation.

The cache management approach is actually determined by the OpenGL driver or host application, and the selection is made through the PCIe interface directly to the Cache Manager. As shown in Table 13, there are four cache management approaches available for selection, but the re-programmability of the FPGA allows the capability for more approaches should they be needed by certain applications. Each cache entry is 32-

bits wide to accommodate one texel color. The texel addresses are not stored in the cache; instead, the texture's base address for each sub-cache is maintained in the Cache Manager and used by the Texel Fetch State Machine.

Table 13. Cache management approaches.

Number of Sub-Caches	Size of Sub-Caches (32-bit Texels)
1	2048
2	1024
4	512
8	256

With the use of any cache, there must be a solid approach for maintaining the validity of the data in the cache. In this application, the cache data is invalidated when the texture data is updated either by the OpenGL driver or the Video Controller. To detect when a cache may be invalid, a dedicated process runs continuously to check for a match in the texture base addresses between the cache textures and the texture being updated. When a match is found, the cache manager flushes all the data from the cache (or sub-cache) and slowly re-reads the texture data to repopulate the cache for further use.

6.3.3 Cache Performance Monitor

The Cache Performance Monitor is responsible for maintaining statistics on the Texture Cache performance such that the OpenGL driver or host application can select the most efficient cache management approach. This feedback is also valuable for performance benchmarking as well as for architectural analysis. The number of cache

hits and misses are captured for each frame, and additional statistics could be implemented in the future to capture the number of used cache elements or the number of cache stalls.

6.3.4 Texture Memory Arbiter

The Texture Memory Arbiter regulates access to and from the external texture memory to maximize pipeline performance. There are possibly up to three sources competing for memory bandwidth. The most critical source is the Texel Fetch State Machine because it has the potential to throttle the performance of the entire GPU pipeline when it stalls. All of the interfaces into and out of the Texture Memory Arbiter have been standardized with Altera's Avalon-MM interface to allow the use of Altera's Multi-Port Front End (MPFE) IP core.

The MPFE core is a weighted round-robin arbiter that supports up to sixteen Avalon-MM ports that are trying to access one external memory through another Avalon-MM port. The core is configurable through the graphical user interface shown in Figures 62 and 63. The number of inputs (slave ports) is three for this application, and the interface to the Texel Fetch State Machine was given 80% of the bandwidth to avoid pipeline stalls. In an actual application, the worst case latency at each port would need to be monitored to validate the bandwidth allocation.

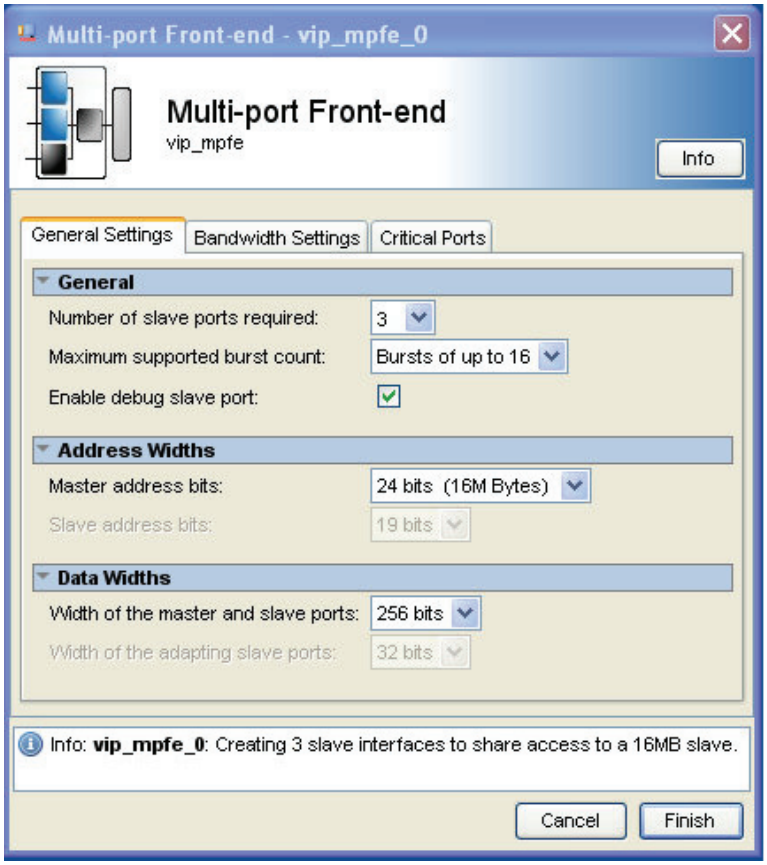


Figure 62: Multi-port Front-end arbiter setup dialog.

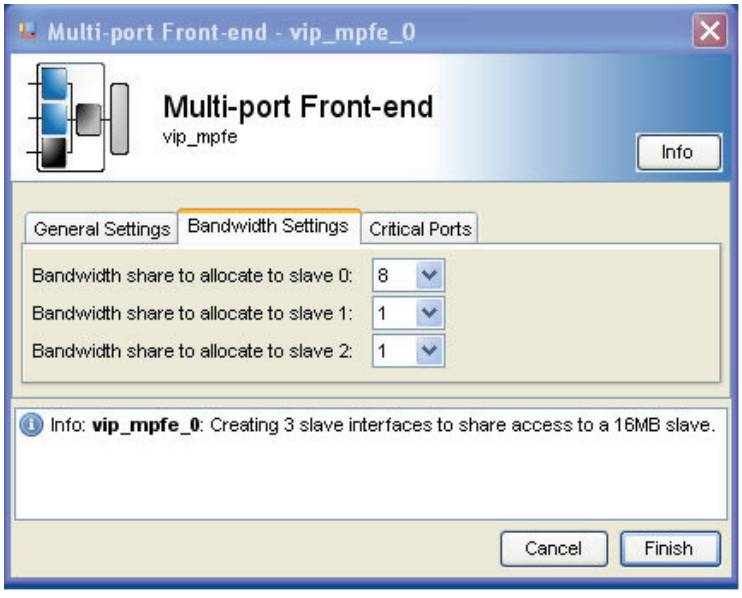


Figure 63: Multi-port Front-end arbiter bandwidth settings.

6.4 Video Controller

The Video Controller provides the ability to receive an external video source and treat its data as a streaming texture. This enhanced feature is beyond the standard scope of OpenGL, but streaming texture functionality could be very useful in many applications. Streaming textures are textures that are updated at a regular rate such as 30 Hz or 60 Hz. The texture can be a full-frame video from a camera source, or it could be as simple as a pattern or color which changes over time. The end result is that a portion of the rendered output appears to have live video or dynamic data embedded within it. The block diagram of the Video Controller architecture is shown in Figure 64.

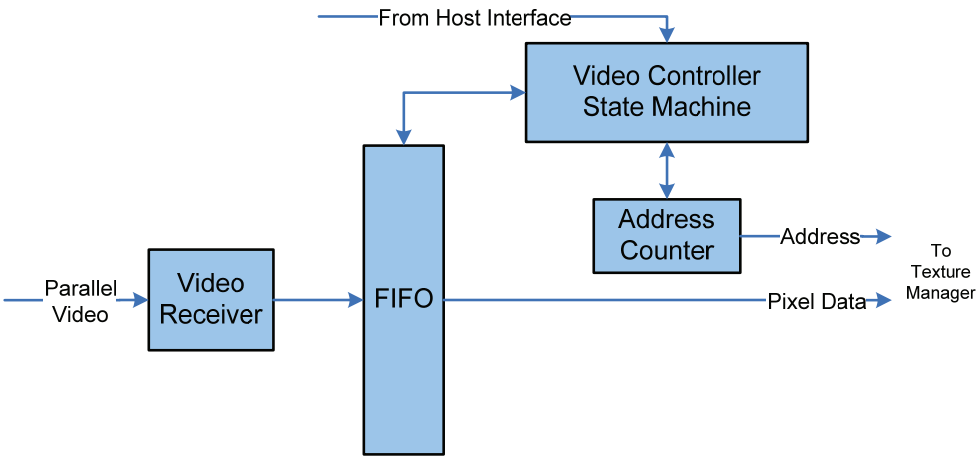


Figure 64: Video Controller block diagram.

The Video Controller is responsible for receiving external video data and transferring it to the texture memory buffer. However, this simple operation requires careful attention due to the dynamic clock boundaries that result from the variable

frequency of the input pixel clock. The solution to this possible timing issue is the creation of a very shallow FIFO buffer whose data can be clocked in with the pixel clock and clocked out with the internal GP clock. The format for the FIFO is shown below in the table below.

Table 14. Video Input FIFO format.

VSYNC	HSYNC	Valid Data Flag	Pixel Data
1 bit	1 bit	1 bit	32 bit
34	33	32	31 0

The Video Receiver module continually writes data into the FIFO at the rising edge of each pixel clock. The data valid bit is set when the external data enable signal is set. In the internal GP clock domain, the Video Controller State Machine continually reads data out of the FIFO. When the data valid bit is set, the pixel data is sent to the Texture Manager along with an address. The address is generated through a counter that increments each time a valid data word is pulled from the FIFO. Thus, sequential addresses are written into the texture memory as the external video is decoded.

6.5 Summary

This chapter proposes an architecture that demonstrates the extendibility of the basic FPGA GPU architecture presented in Chapter 3 of this thesis. Because of its modular architecture, the extended GPU pipeline leverages the existing architectural building blocks from the basic GPU pipeline. The extended GPU pipeline fulfills the comprehensive requirements of OpenGL SC, including vertex processing and fragment

processing. The Vertex Processor, which transforms vertex coordinates and calculates vertex colors, minimizes the FPGA resources required by using state machines to reuse basic mathematical functionality such as matrix multiplication. The Fragment Processor, on the other hand, pipelines the operations to maximize throughput with multiple fragments in-process at one time. Additional GPU capability was also included by developing concepts for texture management and an input video controller. The Texture Manager mitigates the impact of external memory latencies on the GPU pipeline by using a local cache and a credit-based arbitration scheme. The Video Controller accepts parallel RGB video of any resolution and formats it for use as a texture.

CHAPTER 7

FPGA GPU EXTENSIONS FOR VISUALIZATIONS

7.1 Introduction & Previous Work

Scientific visualizations are unique applications that require both data computations and graphical processing to create a visual representation of a scientific phenomenon. Visualizations are widely used to help describe scenarios that would otherwise be too difficult to explain in words. Applications as diverse as bacterial bioluminescence, molecular behavior, and even airport security checkpoints have benefited from visualizations. Previous visualization research was completed in bacterial bioluminescence as well as airport security, and those efforts are briefly described below. The remainder of this chapter provides a motivation for FPGA-based scientific visualizations and then explains the extensions required to the FPGA-based graphics processing architecture.

7.1.1 Airport Security Visualizations

Several visualizations were created as part of a research contract with the National Safe Skies Alliance. Using an OpenGL-based development environment called WorldUp, a visualization tool was created to simulate airport security checkpoints with a three-dimensional perspective. The title of this visualization was Checkpoint Viz, and it allowed the FAA/TSA to model the passenger flow through airport security checkpoints.

The user could select which screening devices would be used at the checkpoint, and a 3D object for each device would be imported into the scene. After some initial input from the user concerning overall passenger traffic, the visualization used statistical models of passenger behavior as well as the throughput of the screening devices to visually represent the passenger flow. The passengers appeared to walk through different paths in the checkpoint with delays at the appropriate areas. Overall the user could monitor the congestion of passenger traffic and modify the selection and placement of security devices to improve throughput.

The Checkpoint Viz tool was also broadened to encompass the entire secured area of the airport facility in Airport Viz. This tool allowed 3D models of entire airports to be inserted into the scene to setup simulations of vehicles as well as pedestrians through various security checkpoints. The National Safe Skies Alliance used both Checkpoint Viz and Airport Viz to help analyze various throughput and security issues on behalf of the FAA and TSA [96].

7.1.2 Vibrio Fischeri

During the development of Checkpoint Viz and Airport Viz, several other researchers at the University of Tennessee also became interested in the capabilities of 3D visualizations. The Biological Modeling Research Group asked for a new tool to visually analyze the bioluminescence of the Vibrio fischeri bacteria. Vibrio fischeri is researched quite often because its bioluminescence properties are affected by quorum sensing. As the quantity of bacteria grows, the luminescence increases as well. Visualization was used to visually investigate how different quorum sensing models

affected the Vibrio fischeri bioluminescence. Since the feedback from the various models was a visual animation of the bacteria, the analysis was very effective [97, 98].

7.2 Motivation for FPGA-based Scientific Visualizations

Scientific visualizations are typically conducted on workstation desktop computers with high-powered graphics processors and multi-core CPUs. These visualizations are capable of producing very realistic visual representations of the scientific algorithms or scenarios, but there can be two main drawbacks to this setup: performance and power. Each of these drawbacks is briefly described below.

The performance of the visualization in a typical desktop computer is limited by three main factors: CPU performance, GPU performance, and algorithmic mapping. The CPU is used for number crunching algorithms that produce OpenGL commands. The GPU converts those OpenGL commands into a visual representation. However, the algorithmic mapping can be the most critical element that affects the visualization performance. If the algorithm is segregated between the CPU and GPU in a manner that does not take advantage of their capabilities, then the performance will be limited. Oftentimes, the GPU hardware is not relied upon enough to calculate vertex transformations and lighting parameters. This functionality is often found in the OpenGL driver, which actually executes in the CPU. The overall performance can improve simply by using the dedicated GPU hardware to perform as much of the graphics work as possible, including vertex transformations, lighting calculations, and primitive culling. Likewise, the CPU is often relied upon to perform complex arithmetic that would run much more efficiently on a GPU as part of a GPGPU effort.

The power consumed by the scientific visualization is directly related to the CPU and GPU hardware. For power-sensitive applications such as remote laboratory experiments or mobile computing, the typical workstation setup is not appropriate. The alternative solution for such applications is simply a lower-powered laptop, but this compromise introduces a significant performance impact. A more satisfactory solution is to architect a computing platform that can accomplish the general-purpose algorithmic calculations as well as the graphical processing in a power-efficient manner.

The typical platform for a visualization is shown in Figure 65. The input data could be as simple as a set of commands at initialization, or it could be a streaming flow of data from various sensors. For example, motion trackers with six degrees-of-freedom can provide coordinate data that can be used to move or rotate the viewpoint in the visualization to create an immersive 3D experience. The sensor data typically goes through a driver and is available to the application running on the operating system. The application also depends upon a GPU driver to handle the OpenGL interface with the GPU. Additional latency and complexity are added at each stage as the data is transferred up to the application and then back down to the hardware.

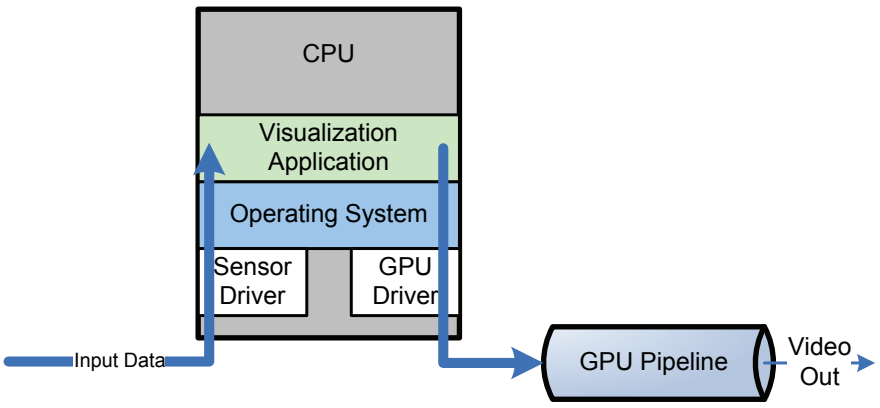


Figure 65: Typical platform for scientific visualizations.

An FPGA-based GPU provides four main advantages to scientific visualizations: latency reduction, integrated processing, customized interfaces, and the capability to extend visualizations to non-desktop environments (mobile, handheld, outdoor). Each of these advantages is briefly described in the following paragraphs.

An FPGA is ideally suited for scientific visualization because it can minimize latencies by having both the computations and the graphics processed in one device without driver, bus, and operating system delays. The integrated processing streamlines the data transfer because the FPGA is designed at a low level without the overhead required to support operating systems and device drivers.

Because the FPGA can be customized for a specific visualization application, unique input interfaces can also be used. Essentially, any input sensor or device could drive the parameters of the visualization once the appropriate sensor interface is developed in the FPGA. Finally, the FPGA's power consumption can be minimized because it can be re-programmed to address the specific computing and graphics needs of the visualization without including any unnecessary features. This minimized power consumption could allow the visualization to be extended to non-desktop environments such as portable, mobile, or embedded applications.

7.3 Extensions Required from FPGA GPU Architecture

To perform scientific visualizations with an FPGA GPU, there are several extensions required from the basic FPGA GPU architecture [99]. Three different

approaches could be taken, mostly depending on the researcher's comfort level in dealing with the details of the algorithmic development. Listed in increasing order of abstraction, the FPGA GPU visualization approaches are:

1. State machine controls data computation data and generates OpenGL commands to the GPU
2. Soft-core processor controls data computation and generates OpenGL commands to the GPU
3. Soft-core processor controls data computation in one GPU core and generates OpenGL commands for second GPU core

These three approaches are described in the following paragraphs.

7.3.1 State Machine Approach

A very simple state machine could be constructed to control the computation of data and the OpenGL interface with the GPU. Figure 66 shows a block diagram of the FPGA. The state machine is not used for any actual computation; rather, it is used just for maintaining the sequence of operations and regulating the data flow through the computation engine. The computation engine is a custom designed piece of firmware that performs the calculations necessary to allow the state machine to make decisions with the OpenGL commands. For example, the visualization algorithm could influence the object colors as the objects move in the scene. The computation engine could be used to calculate the new values of the vertices as well as any lighting changes based upon the algorithm. The vertex data would then be output from the computation engine back to

the state machine. The state machine would then package the data into an OpenGL command for use by the GPU.

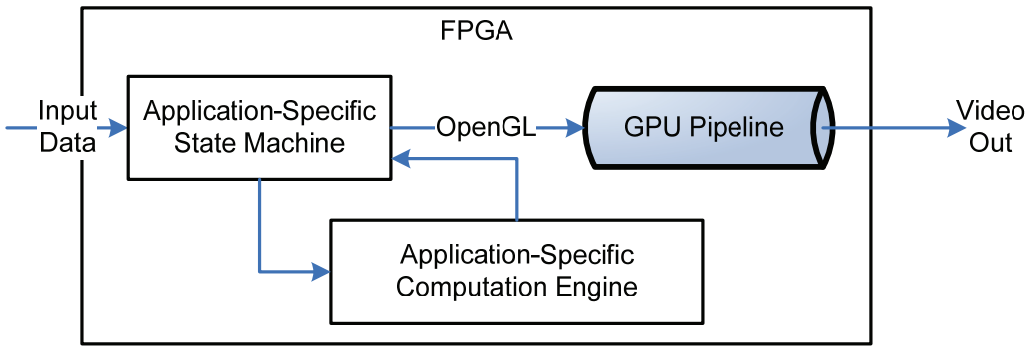


Figure 66: State machine approach.

This approach is the simplest in terms of design complexity because it is targeted for a specific visualization algorithm. The simplicity of this approach also improves its efficiency because there is very little latency introduced to perform the calculations or transfer the OpenGL commands. A similar state machine was developed to test the FPGA GPU architecture during integration, prior to the OpenGL driver development on the Windows PC. This state machine provided a moving set of vertex coordinates along with color changes to exercise the GPU in a standalone fashion. The basic block diagram of this stimulus state machine is shown in Figure 67.

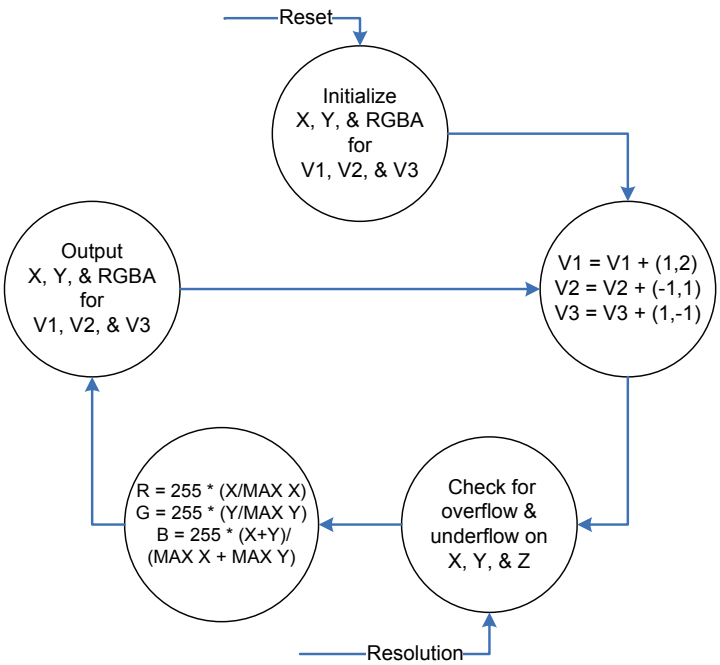


Figure 67: Stimulus state machine used during GPU integration.

7.3.2 Soft-core Processor Approach

If the computations require a bit more sequencing or complexity, then a soft-core processor could be used instead of the state machine, as shown in Figure 68. The soft-core processor could be Altera's Nios II, Xilinx's Microblaze, or a number of other less prevalent processors. In order to maintain a high level of performance, the processor should not actually be in the data computation path. Instead, the arithmetic and algorithmic implementation should be in true hardware where it can be parallelized and optimized with FPGA structures such as embedded multipliers, DSP blocks, and block memory.

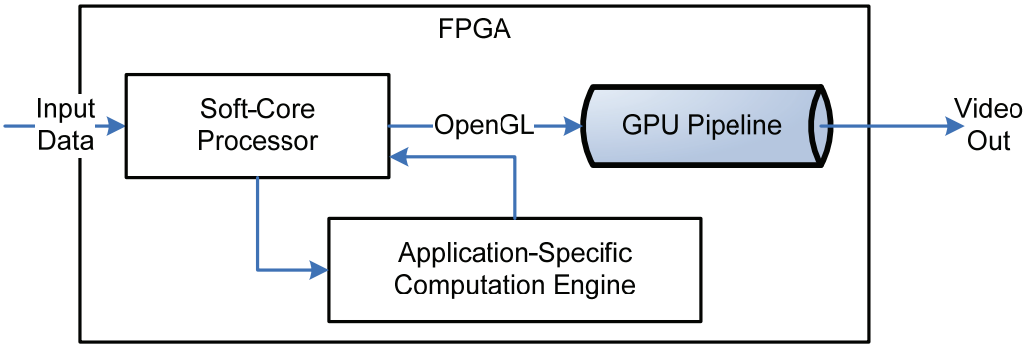


Figure 68: Soft-core processor approach.

This approach provides more flexibility than the state machine due to the easy re-programmability of the soft-core processor. Non-FPGA designers could develop code for the processor as long as the interface to the computation engine and GPU is well understood. The OpenGL interface is well defined, so then the task is just isolated down to making sure that the computation engine has the correct interface. A streaming protocol such as Avalon-ST would be most efficient for the data transfer to and from the computation engine, whereas a simple memory-mapped protocol such as Avalon-MM could be used for the mode settings and control.

7.3.3 Soft-core Processor with Dual GPU pipelines

For the best visualization performance possible with the FPGA GPU architecture, two GPU cores could be used along with a soft-core processor, as shown in Figure 69. The visualization application code running on the soft-core processor would feeds kernels into an OpenCL driver, which then packages the data for use on the first GPU. The computed data is returned to the visualization application code which then uses an

OpenGL driver to issue graphical commands to the second GPU. This second GPU pipeline generates the graphical output for display.

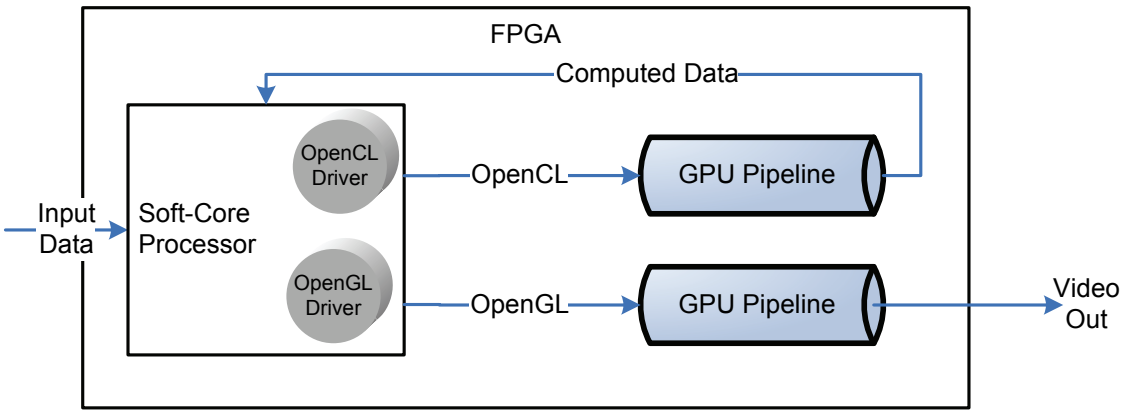


Figure 69: Soft-core processor approach with dual GPU pipelines.

This approach is most efficient for applications that require intensive computations that can benefit from the GPU's computational power. The ability for the GPU to perform general-purpose computations is possible through the OpenCL driver. Once this driver is developed, the GPU core becomes usable to a broader range of general purpose computations – similar to the GPGPU research thrust. Thus, the GPU architecture presented in this thesis would also be applicable to not only scientific visualizations but also to general computational applications.

The dual-GPU approach requires significantly more development time than the previous two approaches because of the need to develop an OpenCL driver for the GPU core. Despite the development time required, this approach also offers the highest level of flexibility to the end user. The visualization algorithm can essentially be written in OpenCL, which has gained widespread industry acceptance for cross-platform

computing. The user is not required to be knowledgeable about the architectural constructs within the GPU core or even the rest of the FPGA-based design. Due to the computation power of the GPU, no new firmware would need to be written to perform the application-specific algorithms in this approach.

7.4 Summary

Visualizations combine algorithmic data computation with graphics processing to provide visual representations for various scenarios found in scientific experiments as well as higher level applications. Previous research in airport traffic visualization and biological luminescence revealed the usefulness of visualizations, and now three approaches have been described for using the FPGA GPU architecture from this thesis to increase performance and to allow the use of visualizations in non-desktop environments.

CHAPTER 8

CONCLUSIONS

8.1 Summary

In this thesis, the research of flexible architecture methods for graphics processing has resulted in an FPGA-based GPU architecture that satisfies many underserved markets. Previous GPU research has produced powerful architectural concepts, but nearly all of the implementations have targeted ASIC structures and capabilities. The continuing improvement of FPGA technology has allowed this focused research effort to result in an FPGA GPU that mitigates device availability concerns while providing a flexible platform for application-specific customizations. The ability to scale the FPGA GPU up or down depending on the application has the potential to reduce power and cost while maximizing performance. This FPGA-based architecture provides a foundation in underserved applications that require low cost, customized features, high reliability, and long device availability.

The FPGA GPU architecture described in this thesis was developed after researching FPGA capabilities as well as existing GPUs. The overall structure of the GPU architecture is motivated by the OpenGL pipeline, with distinct blocks assigned along the pipeline for specific functions. The architecture provides basic graphics functionality that exploits the flexible fabric and unique structures of an FPGA, and it also avoids FPGA limitations in order to optimize performance suitable for many

applications. This architecture is targeted specifically for FPGAs, and each functional block in the modular design was analyzed to best match the FPGA's capabilities with the GPU requirements. The architecture's destination in an FPGA means that its re-programmability can allow customizations and adjustments to permit functionality that was previously not possible in fixed-function pipelines. The architecture is highly modular and scalable to allow its use in a number of different applications.

The development of the FPGA GPU architecture required extensive simulation to identify performance bottlenecks as well as to validate architectural concepts. The iterative process of simulation, analysis, and redesign required a unique simulation approach to accelerate the research. The simulation approach described in Chapter 4 uses a combination of standard tools and custom-developed utilities and scripts to provide a fully automated visual output. Based upon simple text commands, the simulation produces a bitmap file for quick visual analysis as well as a full text file with the raw data for each pixel in the bitmap.

The performance of the FPGA GPU architecture is confirmed with an avionics display application in Chapter 5. The real and immediate need for an FPGA-based graphics processor in the avionics industry was motivated by a discussion on the requirements and current options for avionics graphics processing. A custom circuit card was designed to test the new FPGA GPU, and an existing CPU with existing software was used as the test bench. The avionics implementation of the FPGA GPU shows that after rendering typical avionics graphics, 85% of the GPU's throughput is still available for additional graphics or symbology. This margin represents a 3x speedup in performance while providing a 45% cost reduction, 60% power reduction, 35% circuit

card area reduction, and additional advantages in device availability and certification advantages.

The breadth of this architectural GPU research was demonstrated with a discussion of two extensions that provide either comprehensive graphics processing or scientific visualizations. The architectural extensions were thoroughly discussed to allow future researchers to leverage the foundation presented in this thesis. The applicability of this research for other purposes shows that this research is indeed much more than an FPGA GPU architecture and in fact could be used across a wide range of applications.

8.2 Contributions

8.2.1 FPGA-based GPU Architecture

This research introduced a graphics processing architecture that was developed specifically for an FPGA as opposed to an ASIC implementation. The overall structure of the patent-pending architecture is broken into functional modules with FIFOs separating each module. The FIFOs are used for flow buffering and performance analysis, and they can be implemented in the FPGA's embedded block memory without consuming any logic elements. The choice of multiple FIFOs in the architectural structure was made because of the capabilities of the FPGA technologies. In addition, each of the functional modules was specifically designed based upon the FPGA's capabilities. In each module, specific FPGA components such as DSP blocks, soft-core multi-threaded processors, and dual-clocked FIFOs were all used to accomplish the required functionality. The design of each module also accounted for the FPGA's

inherent disadvantage of routing delays by using pipelining and synchronous design techniques.

Previous GPU architectures that were developed for ASIC implementations most often relied on extremely high clock rates, massively parallel ALUs, and significant embedded low-latency memory. Because of FPGA limitations, these ASIC architectures cannot be targeted towards an FPGA without a significant decrease in performance. Previous research into FPGA-based GPUs has mostly resulted in soft-core processor implementations that lack the performance capabilities or scalability required for a diverse set of applications. The GPU architecture presented in this thesis provides an FPGA-focused design with scalability and performance suitable for many industrial or embedded applications. The use of an FPGA as the final target for the architectural design also mitigates the risk of device obsolescence and allows the FPGA to be crafted specifically for the application requirements.

8.2.2 Rasterizer Architecture

The second contribution that this research provides is a patent-pending Rasterizer architecture that uses a multi-threaded soft-core processing approach. There are many possible techniques for rasterizing lines in an FPGA, but this approach is unique in that it is highly scalable to support nearly eight times its performance with very little impact to the design. The Rasterizer architecture as well as its scalability is discussed in Chapter 3.

The rasterization engine exploits a high-performance packet processing engine from Altera that uses a task/event relationship between a multithreaded soft processor and hardware acceleration blocks. Although this engine has previously only been used in

packet processing applications, this FPGA-based GPU pipeline provides a unique set of requirements that make the packet processing perspective appropriate for line rasterization. A custom rasterizing hardware module could have been developed for this architecture, but doing so would have limited the versatility and scalability of the architecture. Furthermore, the migration of the packet processing platform from its intended networking application into a graphics rendering application provides novelty to this research.

8.2.3 Simulation Approach

The third contribution of this research is the unique simulation approach developed to accelerate the analysis of FPGA-based processing. This approach, which is described in Chapter 4, uses a combination of standard simulation tools along with custom developed utilities to provide a fully automated visual output based upon the FPGA design. For each simulation, the output is presented not only in a waveform but also in a text file and a visual bitmap file. The architectural research in this thesis would not have been possible without the efficiency and accuracy provided by the automated visual simulation. This same simulation approach could be leveraged for many other research areas such as video filtering or sensor fusion.

A TXT2BMP software utility was developed to aid in the automated visual simulation. As pixels are written into the frame buffer, the simulation test bench writes the pixel data to a text file. The TXT2BMP software then converts this pixel text file into a bitmap that can be viewed at the conclusion of the simulation.

8.2.4 FPGA GPU Extendibility

The final contribution of this research is the extendibility of the FPGA-based graphics processing architecture to support applications more broad than just basic graphics processing. Chapter 6 describes the architectural extensions for supporting comprehensive graphics processing that includes vertex processing, fragment processing, and texture management. Chapter 7 explains how this FPGA-based graphics processing architecture can also be leveraged in general-purpose computations or scientific visualizations, which combine computations with graphics processing. Together, these extensions demonstrate that this FPGA-based graphics processing architecture has a much broader impact than just basic graphics processing.

A key characteristic of the extended FPGA GPU architecture in this thesis is that all of the vertex and fragment processing is completed with true hardware acceleration. Many embedded or mobile applications depend on the OpenGL driver to transform the coordinates and perform the lighting calculations as a part of the vertex processing. Alternately, this extended FPGA GPU provides the entire OpenGL SC pipeline with full hardware acceleration.

The FPGA GPU architectural extensions to support vertex processing and fragment processing allow the GPU to be used in a much broader range of applications. The vertex processing provides a three-dimensional perspective to the graphics, and the fragment processing permits more complex lighting and texturing effects. The end result of this additional functionality is that the graphical display appears much more realistic. As such, the extended FPGA GPU architecture can be used in more consumer devices where aesthetically pleasing displays are more important. As an alternative, these

advanced features could also be used in industrial or avionics applications to provide better situational awareness.

The use of GPUs for general purpose computations or scientific visualizations is not a novel idea. However, the FPGA GPU architecture in this thesis provides unique advantages to the accelerations of scientific visualizations. Three approaches were described in Chapter 7 to leverage the FPGA GPU to extend the applicability of scientific visualizations to non-desktop environments. Each of the approaches takes advantage of the FPGA's programmable fabric to perform application-specific computations that feed into the GPU for the visualization.

8.3 Future Work

In the near-term future, the avionics application of the FPGA GPU technology that was described in Chapter 5 could be optimized based upon the results so far. Further research could either target a smaller FPGA device with less logic resources to reduce power and cost, or add functionality such as video overlay to the existing FPGA device for a more comprehensive solution.

In addition, the extended FPGA GPU architecture discussed in Chapter 6 could be fully implemented on a custom circuit card. This extension would allow a more complex set of graphical functionality but also requires a full OpenGL software driver. The driver would not need to implement any of the GPU pipeline tasks, but it would be responsible for operating system interfaces, memory allocation, and OpenGL command formatting.

For research long-term, the scientific visualization approaches discussed in Chapter 7 could be further analyzed and implemented. To maximize the efficiency of the

FPGA GPU, a visualization application that does not require advanced user interaction or an operating system would be best for initial research. In fact, the user input could be as simple as push buttons that are read by the FPGA. Overall the goal would be to validate the usefulness of this FPGA GPU architecture is non-traditional applications such as scientific visualizations.

Appendix A - Vincent SC Test Application

The following code was compiled together with the Vincent SC codebase as downloaded from Sourceforge. SDL also needs to be installed on the same PC in order to provide the windowing capabilities.

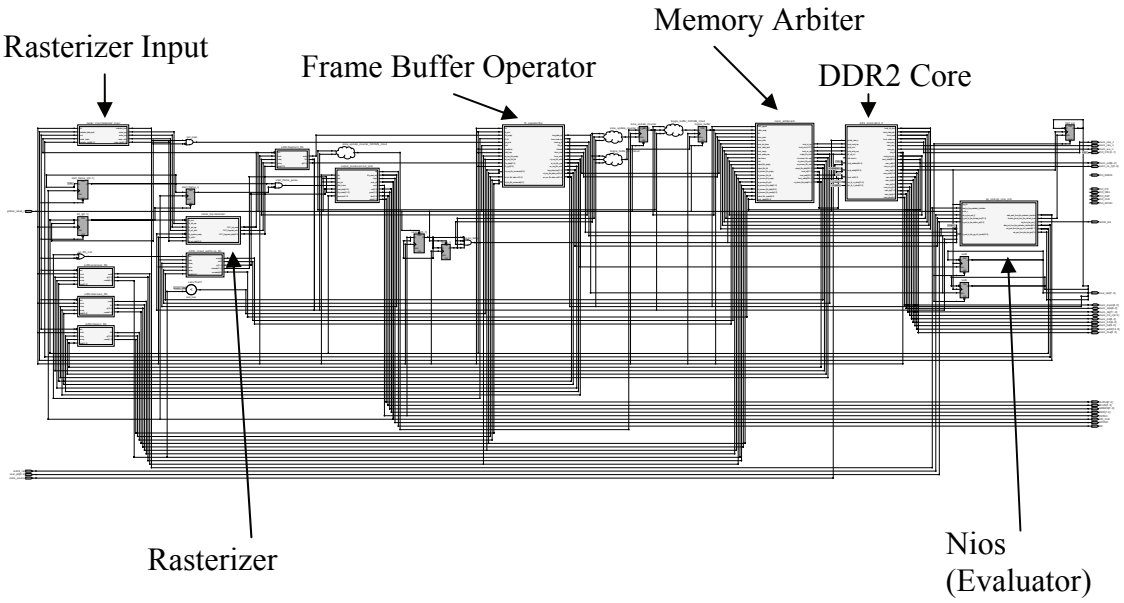
This code produces the shaded triangle that is shown in Figure 9.

```
#include <SDL.h>
#include "GL/gl.h"
#include "GL/vgl.h"

#include <stdlib.h>
#include <string.h>

#define     WINDOW_WIDTH      640
#define     WINDOW_HEIGHT     480

/* Re-introduce 32-bit word encoding of 32-bit RGBA values */
#define GL_UNSIGNED_INT_8_8_8_8           0x8035
#define GL_UNSIGNED_INT_8_8_8_8_REV       0x8367

/* The number of our GLUT window */
SDL_Surface *sdlSurface;
VGL_Surface glSurface;

int width = WINDOW_WIDTH;
int height = WINDOW_HEIGHT;

/**********************************************************************
**********************
*   DrawTriangle
*   Desc: triangle
**********************************************************************
**********************/
void DrawTriangle( )
{
    float vertices[4*3] =
      { -0.5f,-0.5f, 0.0f,
         0.0f, 0.5f, 0.0f,
         0.5f,-0.5f, 0.0f,
         0.0f,-1.0f,0.0f,
      };
    float colors[4*4] =
      {  1.0f, 0.0f, 1.0f, 1.0f,                          // Red
         0.0f, 1.0f, 0.0f, 1.0f,                          // Green
         0.0f, 0.0f, 1.0f, 1.0f,                          // Blue
         1.0f, 0.0f, 1.0f, 1.0f,                          // Purple
      };
```

```
    glShadeModel(GL_SMOOTH);              // Enable Smooth Shading
    glClearColor(1.0f, 1.0f, 1.0f, 0.5f); // Black Background

    glClear(GL_COLOR_BUFFER_BIT);         // Clear Screen & Depth Buff

    glLoadIdentity();                     // Reset the Modelview Matrix

    glOrthof(-1.0f,1.0f,-1.0f,1.0f,-1.0f,1.0f); // modeling coords

    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);
    glColorPointer(4, GL_FLOAT, 0, (const void *)&colors[0]);
    glVertexPointer(3, GL_FLOAT, 0, (const void *)&vertices[0]);

    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

    glDisableClientState(GL_COLOR_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);
}


#define ESCAPE 27

int main(int argc, char* argv[])
{
      SDL_Event event;

      SDL_Init(SDL_INIT_VIDEO);
      atexit(SDL_Quit);
      vglInitialize();

      sdlSurface = SDL_SetVideoMode(width, height, 0, SDL_ANYFORMAT);
      glSurface = vglCreateSurface(width, height, GL_RGBA,
      GL_UNSIGNED_INT_8_8_8_8_REV, 0);
      vglMakeCurrent(glSurface, glSurface);

      for (;;) {
            SDL_PollEvent(&event);

            if (event.type == SDL_QUIT) {
                  break;
            }

            DrawTriangle();
            vglSwapBuffers(sdlSurface, glSurface);
      }

      return 0;
}
```

Appendix B – GPU Overall Block Diagram

The block diagram of the overall FPGA GPU shown below was created by the RTL Viewer in Altera's Quartus2 development environment. It shows how the blocks are interconnected after the Analysis & Synthesis tools have completed processing the VHDL code.



Appendix C – Evaluator Source Code

The following source code was developed to implement the Evaluator's functionality in the Nios2 CPU, as described in Chapter 3.

```
/*
 * Description
 * *************
 */

#include "top.h"

/* Declare one global variable to capture the output of the buttons
(SW0-SW3),
 * when they are pressed.
 */

volatile int switch_edge_capture;
volatile int binary_mode = 0;
int ctrl_edge_capture = 16;
volatile int starting_bmp_mode = 0;
volatile int bmp_mode = 0;
volatile int pixel_counter = 0;
volatile int max_pixel_count = 307200; //VGA
volatile int capture_fill_timer = 0;
volatile int capture_finish_timer = 0;
double elapsed_time = 0;
int timer_lower = 0;
int timer_upper = 0;
int display_list[8][97] = {{0}};    //8 display lists with 32 commands
each
int num_display_lists = 0;
int num_items_in_display_list = 0;
int display_list_call[8] = {0};

/*********************************************************************
 * static void MenuItem( char letter, char *string )
 *
 * Function to define a menu entry.
 *  - Maps a character (defined by 'letter') to a description string
 *    (defined by 'string').
 *

*********************************************************************/

static void MenuItem( char letter, char *name )
{
  printf("     %c:  %s" ,letter, name);
}

/*****************************************************************
*  Function: GetInputString
*
*  Purpose: Parses an input string for the character '\n'.  Then
*           returns the string, minus any '\r' characters it
```

```
*           encounters.
*
*****************************************************************/
void GetInputString( char* entry, int size, FILE * stream )
{
  int i;
  int ch = 0;

  for(i = 0; (ch != '\n') && (i < size); )
  {
    if( (ch = getc(stream)) != '\r')
    {
      entry[i] = ch;
      i++;
    }
  }
}

/*****************************************************************************
*********
 *
 * static void DoUtilityMenu( void )
 *
 * Generates the GP Utility menu.
 *

*****************************************************************************
*******/

static void DoUtilityMenu( void )
{

    //int ctrlreg;

  while(!starting_bmp_mode)
  {
    printf("-----------------\r\n");
    printf("FPGA-GP Board Diagnostics");
    printf(" GP Utility Menu\r\n");
    MenuItem('b', "Capture BMP\r\n");
    MenuItem('v', "Switch to VGA\r\n");
    MenuItem('x', "Switch to XGA\r\n");
    MenuItem('p', "Portrait\r\n");
    MenuItem('l', "Landscape\r\n");
    MenuItem('f', "60Hz Frame Rate\r\n");
    MenuItem('s', "30Hz Frame Rate\r\n");
    printf("     q:  Exit\r\n");
    printf("Select Choice (v-l): [Followed by <enter>]\r\n");

    static char ch;
    static char entry[4];
    GetInputString( entry, sizeof(entry), stdin );
    if(sscanf(entry, "%c\n", &ch))
    {
        if( ch >= 'A' && ch <= 'Z' )
        ch += 'a' - 'A';
        if( ch == 27 )
```

```
        ch = 'q';
    }

    switch(ch)
    {
      case 'b':
        printf( "Starting BMP Capture... data is incoming!\r\n" );
        starting_bmp_mode = 1;
        break;

      case 'v':
        printf( "VGA Mode selected.\r\n" );
        max_pixel_count = 307200;
        // CLEAR XGA_VGAN BIT
        IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x4);

        break;

      case 'x':
        printf( "XGA Mode selected.\r\n" );
        max_pixel_count = 786432;
        // SET XGA_VGAn BIT
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x4);

        break;

      case 'p':
        printf( "Portrait Mode selected.\r\n" );
        // SET PORT_LANDn BIT
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x8);
        break;

      case 'l':
        printf( "Landscape Mode selected.\r\n" );
        // CLEAR PORT_LANDn BIT
        IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x8);
        break;

      case 'f':
        printf( "60 Hz frame rate selected.\r\n" );
        //# of 50MHz clock ticks in 60Hz period = 2500000
        IOWR_ALTERA_AVALON_TIMER_PERIODL(REFRESH_TIMER_BASE,0x25A0);
        IOWR_ALTERA_AVALON_TIMER_PERIODH(REFRESH_TIMER_BASE,0x26);
        IOWR_ALTERA_AVALON_TIMER_CONTROL(REFRESH_TIMER_BASE,0x6);
//ENABLE REFRESH TIMER
        break;

      case 's':
        printf( "30 Hz frame rate selected.\r\n" );
        //# of 50MHz clock ticks in 30Hz period = 5000000
        IOWR_ALTERA_AVALON_TIMER_PERIODL(REFRESH_TIMER_BASE,0x4B40);
        IOWR_ALTERA_AVALON_TIMER_PERIODH(REFRESH_TIMER_BASE,0x4C);
        IOWR_ALTERA_AVALON_TIMER_CONTROL(REFRESH_TIMER_BASE,0x6);
//ENABLE REFRESH TIMER
        break;

    }
```

```
    if ( ch == 'q' )
    {
      break;
    }
  }
}

/*****************************************
 * GP CTRL PIO Functions
 *****************************************/
FILE *bmp_port;
int bitmap_pixel;
int received_bmp_data;

/*****************************************************************
 * static void handle_ctrl_interrupts( void* context, alt_u32 id)*
 *                                                               *
 * Handle interrupts from the GP ctrl i/o.                       *
 * This handler sets *context to the value read from the button  *
 * edge capture register.  The ctrl edge capture register        *
 * is then cleared and normal program execution resumes.         *
 * The value stored in *context is used to control program flow  *
 * in the rest of this program's routines.                       *
 *****************************************************************/

static void handle_ctrl_interrupts(void* context, alt_u32 id)
{
  /* Cast context to edge_capture's type.
   * It is important to keep this volatile,
   * to avoid compiler optimization issues.
   */

  //volatile int* ctrl_edge_capture_ptr = (volatile int*) context;
  /* Store the value in the Button's edge capture register in *context.
*/
  ctrl_edge_capture =
IORD_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE);

  if (ctrl_edge_capture & 1)  // vsync
  {
    // clear interrupt
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE, 0x1);
    IORD_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE); //An extra
read call to clear of delay through the bridge

    IOWR_ALTERA_AVALON_TIMER_PERIODL(DIAGNOSTIC_TIMER_BASE,0xFFFF);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(DIAGNOSTIC_TIMER_BASE,0xFFFF);
    IOWR_ALTERA_AVALON_TIMER_CONTROL(DIAGNOSTIC_TIMER_BASE,0x6);
//RESET DIAGNOSTIC TIMER

    if (starting_bmp_mode)
    {
        starting_bmp_mode = 0;
        bmp_mode = 1;

        // Enable interrupt on data_ready.
```

```
        IOWR_ALTERA_AVALON_PIO_IRQ_MASK(GP_CTRL_INPUTS_BASE, 0x4);

        // PULSE START_BMP HIGH
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x100);
        IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x100);

        // PULSE GET_PIXEL HIGH
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x200);
        IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x200);

    }
    else if (!bmp_mode)
    {

        // PULSE FILL_FB HIGH
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x10);
        IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x10);
    }

  }
  else if (ctrl_edge_capture & 2)   // update rate has dropped to 30Hz
  {
    // clear interrupt
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE, 0x2);
    IORD_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE); //An extra
read call to clear of delay through the bridge

  }
  else if (ctrl_edge_capture & 4)
    {
        //bitmap data ready pulse
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE, 0x4);
        IORD_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE); //An
extra read call to clear delay through the bridge

     if ((bmp_mode) && (pixel_counter < max_pixel_count))
        {
            received_bmp_data = 1;
            pixel_counter++;
        }
        else if (bmp_mode)
        {
            // max pixel count has been reached - bmp capture complete
            bmp_mode = 0;
            pixel_counter = 0;
            printf("bitmap capture is complete\r\n");
            // Enable interrupt on vsync & refresh_pulse.
            IOWR_ALTERA_AVALON_PIO_IRQ_MASK(GP_CTRL_INPUTS_BASE, 0x3);
        }
    }
  else if (ctrl_edge_capture & 8)
    {
        // finished interrupt
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE, 0x8);
        IORD_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE); //An
extra read call to clear delay through the bridge
```

```
        IOWR_ALTERA_AVALON_TIMER_SNAPL (DIAGNOSTIC_TIMER_BASE, 0);
        timer_lower =
IORD_ALTERA_AVALON_TIMER_SNAPL(DIAGNOSTIC_TIMER_BASE) &
ALTERA_AVALON_TIMER_SNAPL_MSK;
        timer_upper =
IORD_ALTERA_AVALON_TIMER_SNAPH(DIAGNOSTIC_TIMER_BASE) &
ALTERA_AVALON_TIMER_SNAPH_MSK;

        elapsed_time = (0xFFFFFFFF - ((timer_upper << 16) |
timer_lower));
        capture_finish_timer = 1;
    }

  else if (ctrl_edge_capture & 16)
    {
        // clear frame buffer complete
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE, 0x10);
        IORD_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE); //An
extra read call to clear delay through the bridge

        if (capture_fill_timer)
        {
            IOWR_ALTERA_AVALON_TIMER_SNAPL (DIAGNOSTIC_TIMER_BASE, 0);
            timer_lower =
IORD_ALTERA_AVALON_TIMER_SNAPL(DIAGNOSTIC_TIMER_BASE) &
ALTERA_AVALON_TIMER_SNAPL_MSK;
            timer_upper =
IORD_ALTERA_AVALON_TIMER_SNAPH(DIAGNOSTIC_TIMER_BASE) &
ALTERA_AVALON_TIMER_SNAPH_MSK;

            elapsed_time = (0xFFFFFFFF - ((timer_upper << 16) |
timer_lower));
        }

        // search for any display lists and send them to FIFO
        int m = 0;
        int i = 0;
        while (m < 8)
        {
            if (display_list_call[m] == 1)
            {
                i = 1;
                while (i<96)
                {
                     if (display_list[m][i] != 0)
                     {

IOWR_ALTERA_AVALON_FIFO_DATA(EVAL_FIFO_MM_IN_BASE,display_list[m][i]);

IOWR_ALTERA_AVALON_FIFO_DATA(EVAL_FIFO_MM_IN_BASE,display_list[m][i+1])
;

IOWR_ALTERA_AVALON_FIFO_DATA(EVAL_FIFO_MM_IN_BASE,display_list[m][i+2])
;
                     }

                     i+=3;
```

```
                }
            }
            m++;

        }



    }

  else printf("unexpected interrupt: %i\r\n",ctrl_edge_capture);


}

/* Initialize the CTRL PIO. */

static void init_ctrl_pio()
{
  /* Recast the edge_capture pointer to match the alt_irq_register()
function
  * prototype. */
  void* ctrl_edge_capture_ptr = (void*) &ctrl_edge_capture;
  // Set output direction of specific bits

  //Clear all output bits
  IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x3FF);

  // Enable interrupt on vsync & refresh_pulse.
  IOWR_ALTERA_AVALON_PIO_IRQ_MASK(GP_CTRL_INPUTS_BASE, 0x1F);
  // Reset the edge capture register.
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(GP_CTRL_INPUTS_BASE, 0x0);
  // Register the interrupt handler.
  alt_irq_register(GP_CTRL_INPUTS_IRQ, ctrl_edge_capture_ptr,
handle_ctrl_interrupts);
}

/* Tear down the ctrl_pio. */
static void disable_ctrl_pio()
{
  /* Disable interrupts from the ctrl PIO component. */
  IOWR_ALTERA_AVALON_PIO_IRQ_MASK(GP_CTRL_INPUTS_BASE, 0x0);
  /* Un-register the IRQ handler by passing a null handler. */
  alt_irq_register( GP_CTRL_INPUTS_IRQ, NULL, NULL );
}


/*********************************************************************
*********
 *
 * static void DoFillFBMenu( void )
 * Generates the Fill Frame Buffer menu.
 *
*********************************************************************
*******/

static void DoFillFBMenu( void )
```

```
{
  volatile alt_u8 led;

  while(1)
  {
    printf("-----------------\r\n");
    printf("FPGA-GP Board Diagnostics");
    printf(" Fill Frame Buffer Menu\r\n");
    MenuItem('r', "RED\r\n");
    MenuItem('b', "BLUE\r\n");
    MenuItem('g', "GREEN\r\n");
    MenuItem('w', "WHITE\r\n");
    MenuItem('k', "BLACK\r\n");
    printf("     q:  Exit\r\n");
    printf("Select Choice (r-k): [Followed by <enter>]\r\n");

    static char ch;
    static char entry[4];
    GetInputString( entry, sizeof(entry), stdin );
    if(sscanf(entry, "%c\n", &ch))
    {
        if( ch >= 'A' && ch <= 'Z' )
        ch += 'a' - 'A';
        if( ch == 27 )
        ch = 'q';
    }

    if (capture_fill_timer)
    {
        printf("upper = %u, lower = %u, elapsed time =
%.1f\r\n",timer_upper, timer_lower, elapsed_time);
        capture_fill_timer = 0;
    }


    switch(ch)
    {

      case 'r':
        /* Turn the red LED on along with the FB Fill discrete. */
        led = 0xc;
        IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x20);
        capture_fill_timer = 1;
        printf("Fill RED\r\n");
        break;

      case 'b':
        /* Turn the blue LED on along with the FB Fill discrete. */
        led = 0x9;
        IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x80);
        printf("Fill BLUE\r\n");
        break;

      case 'g':
        /* Turn the green LED on along with the FB Fill discrete. */
```

```
        led = 0xa;
        IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x40);
        printf("Fill GREEN\r\n");
        break;

      case 'w':
        /* Turn the white LED on along with the FB Fill discrete. */
        led = 0x8;
        IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0xE0);
        printf("Fill WHITE\r\n");
        break;

      case 'k':
        /* Turn the black LED on along with the FB Fill discrete. */
        led = 0xf;
        IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);
        IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0xE0);
        printf("Fill BLACK\r\n");
        break;
    }

    if ( ch == 'q' )
    {
      /* Turn off all LEDs along with the FB Fill discrete. */
        led = 0xFF;
        IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);
      break;
    }
  }
}



/*****************************************************************************
*********
 *
 * Generates the top level menu for this diagnostics program.
 *
*****************************************************************************
*******/

static void ProcessMenus( void )
{
  /* Output the top-level menu to STDOUT */
  static char ch;
  static char entry[4];

  while ((!starting_bmp_mode)&&(!bmp_mode))
  {
    printf("-----------------\r\n");
    printf("FPGA-GP Board Diagnostics");
    printf(" Main Menu\r\n");
    MenuItem( 'a', "Test LEDs\r\n");
    MenuItem( 'b', "Enter Binary Mode\r\n");
    MenuItem( 'c', "Button/Switch Test\r\n");
```

```
    MenuItem( 'f', "Fill Frame Buffer\r\n");
    MenuItem( 'u', "Utility Menu\r\n");

    printf("     q:  Exit\r\n");

    // get character from stdin
    GetInputString( entry, sizeof(entry), stdin );
    if(sscanf(entry, "%c\n", &ch))
    {
        if( ch >= 'A' && ch <= 'Z' )
        ch += 'a' - 'A';
        if( ch == 27 )
        ch = 'q';
    }

    // process received character
    switch(ch)
    {
      MenuCase('a',TestLEDs);
      case 'b':
            printf("Entering binary mode...\r\n");
            binary_mode = 1;
            //fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);
            break;
      MenuCase('c',TestButtons);
      MenuCase('f',DoFillFBMenu);
      MenuCase('u',DoUtilityMenu);
      case 'q': break;
      default:  printf(" -ERROR: %c is an invalid entry.  Please try
again\r\n", ch); break;
    }



    if ((ch == 'q' )||(binary_mode))
      break;



  }
  return;
}

/*****************************************************************
*********
 *
 * Processes the binary inputs and acts based upon interrupt events
 *
*****************************************************************
*******/
int num_bytes_rcvd = 0;
static int packet[10];
alt_u32 pipeline_cmd_data[3];
int creating_display_list = 0;
int cmd_ptr;

static void ProcessBinary( void )
```

```
{
  //printf("processing in binary now\n");

  alt_u32 pipeline_word;
  int ctrlreg = 0;
  int list_ptr, line_ptr, i;

  //while (num_bytes_rcvd < 10)
  //{
    //if (scanf("%x",&n) != 0)
    //if (_getc_nolock(stdin) != 1)
    //{
        packet[num_bytes_rcvd] = getc(stdin);
        //if (packet[num_bytes_rcvd] != NULL)
        //{
            printf("%x",packet[num_bytes_rcvd]);
            num_bytes_rcvd++;
        //}

    //}
    int good_packet = 0;
    if (num_bytes_rcvd == 10)
    {
      num_bytes_rcvd = 0;
      if (packet[0] == 0xF0)
      {
        printf(" Eval | New Display List ID: %x\n",packet[1]);
        if (num_display_lists < 8)
        {
            int k = 0;
            while (display_list[k][0] != 0)
            {
                k++;
            }
            //printf("found it, assigning new DL to %i\n",k);

            list_ptr = (int*) malloc(3*sizeof(alt_u32));
            if (list_ptr == NULL)
            {
                printf("ERROR");
            }
            else
            {
                creating_display_list = k+1;
                list_ptr = packet[1];
                num_display_lists++;
                display_list[creating_display_list-1][0] = packet[1];
                num_items_in_display_list = 1;
            }
        }
        else
        {
            printf("ERROR - no display list memory available\n");
        }
        //creating_display_list = packet[1];

      }
```

```
      else if (packet[0] == 0xFF)
      {
        printf(" Eval | End Display List ID: %x\n",packet[1]);
        creating_display_list = 0;
      }
      else if (packet[0] == 0xF5)
      {
        printf(" Eval | Call Display List ID: %x\n",packet[1]);
        int j = 0;
        while (j < 8)
        {
            //printf("id = %i",display_list[j][0]);
            if (display_list[j][0] == packet[1])
            {
                display_list_call[j] = 1;
                printf("found display list at j = %i\n",j);
                i = 1;
                while (i<97)
                {
                    if (display_list[j][i] != 0)
                    {
                        cmd_ptr = display_list[j][i];
                        printf("%x %x %x, ",display_list[j][i],
display_list[j][i+1], display_list[j][i+2]);
                    }

                    i+=3;
                }
                printf("\n");
            }


            j++;
        }
      }
      else if (packet[0] == 0xF8)
      {
        printf(" Eval | Stop Display List ID: %x\n",packet[1]);
        int j = 0;
        while (j < 8)
        {
            //printf("id = %i",display_list[j][0]);
            if (display_list[j][0] == packet[1])
            {
                display_list_call[j] = 0;
                printf("found display list at j = %i\n",j);
            }
            j++;
        }


      }
      else if (packet[0] == 0xFD)
      {
        printf(" Eval | Delete Display List ID: %x\n",packet[1]);
        int j = 0;
        while (j < 8)
```

```
        {
            if (display_list[j][0] == packet[1])
            {
                i = 0;
                while (i<97)
                {
                    display_list[j][i]=0;  //clear all display lists
                    i++;
                }
                display_list_call[j] = 0;
                num_display_lists--;
                //printf("display_list_call = %i\n",display_list_call);
                //printf("\n");
            }
            j++;
        }
      }
      else if ((packet[0] >= 0x80) && (packet[0] < 0x90))
      {
        if (creating_display_list)
        {
            printf(" Saving to DL %i: ",creating_display_list-1);
            line_ptr = (int *) malloc(3*sizeof(alt_u32));
            if (line_ptr == NULL) printf("ERROR");
            pipeline_cmd_data[0] = (packet[0]*256) + packet[1];
            pipeline_cmd_data[1] = (packet[2]*16777216) +
(packet[3]*65536) + (packet[4]*256) + packet[5];
            pipeline_cmd_data[2] = (packet[6]*16777216) +
(packet[7]*65536) + (packet[8]*256) + packet[9];
            line_ptr = pipeline_cmd_data[0];
            display_list[creating_display_list-
1][num_items_in_display_list] = line_ptr;
            num_items_in_display_list++;
            line_ptr = pipeline_cmd_data[1];
            display_list[creating_display_list-
1][num_items_in_display_list] = line_ptr;
            num_items_in_display_list++;
            line_ptr = pipeline_cmd_data[2];
            display_list[creating_display_list-
1][num_items_in_display_list] = line_ptr;
            num_items_in_display_list++;
            printf("\n");
        }
        else
        {
            printf(" Pipeline Data: ");
            pipeline_word = (packet[0]*256) + packet[1];

IOWR_ALTERA_AVALON_FIFO_DATA(EVAL_FIFO_MM_IN_BASE,pipeline_word);
            printf("%x ",pipeline_word);
            pipeline_word = (packet[2]*16777216) + (packet[3]*65536) +
(packet[4]*256) + packet[5];

IOWR_ALTERA_AVALON_FIFO_DATA(EVAL_FIFO_MM_IN_BASE,pipeline_word);
            printf("%x ",pipeline_word);
            pipeline_word = (packet[6]*16777216) + (packet[7]*65536) +
(packet[8]*256) + packet[9];
```

```
IOWR_ALTERA_AVALON_FIFO_DATA(EVAL_FIFO_MM_IN_BASE,pipeline_word);
            printf("%x ",pipeline_word);
            printf("\n");
        }
      }
      else if (packet[0] == 0xB0)
      {
        printf(" Frame Buffer Control: ");
        if (packet[2] == 0x00)
        {
            printf("Landscape VGA ");
            max_pixel_count = 307200;

            // CLEAR BITS 3-2-1

IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0xE);

        }
        else if (packet[2] == 0x01)
        {
            printf("Portrait VGA ");
            max_pixel_count = 307200;

            // CLEAR BITS 2-1
IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x6);
            // SET BIT 3
            IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x8);

        }
        else if (packet[2] == 0x0F)
        {
            printf("Landscape SVGA ");
            max_pixel_count = 480000;
            // CLEAR BITS 3-2

IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0xC);
            // SET BIT 1
            IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x2);

        }
        else if (packet[2] == 0x1F)
        {
            printf("Portrait SVGA ");
            max_pixel_count = 480000;
            // CLEAR BIT 2
IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x4);
            // SET BITS 3 & 1
            IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0xA);

        }
        else if (packet[2] == 0xF0)
        {
            printf("Landscape XGA ");
            max_pixel_count = 786432;

            // SET BIT 2
```

```
            IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x4);
            // CLEAR BITS 3 & 1

IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0xA);
        }
        else if (packet[2] == 0xF1)
        {
            printf("Portrait XGA ");
            max_pixel_count = 786432;

            // SET BITS 3-2
            IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0xC);
            // CLEAR BIT 1

IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x2);
        }

        if (packet[3] == 0x00)
        {
            printf("60 Hz ");
            //# of 50MHz clock ticks in 60Hz period = 2500000

IOWR_ALTERA_AVALON_TIMER_PERIODL(REFRESH_TIMER_BASE,0x25A0);
            IOWR_ALTERA_AVALON_TIMER_PERIODH(REFRESH_TIMER_BASE,0x26);
            IOWR_ALTERA_AVALON_TIMER_CONTROL(REFRESH_TIMER_BASE,0x6);
//ENABLE REFRESH TIMER
        }
        else if (packet[3] == 0x0F)
        {
            printf("30 Hz ");
            //# of 50MHz clock ticks in 30Hz period = 5000000

IOWR_ALTERA_AVALON_TIMER_PERIODL(REFRESH_TIMER_BASE,0x4B40);
            IOWR_ALTERA_AVALON_TIMER_PERIODH(REFRESH_TIMER_BASE,0x4C);
            IOWR_ALTERA_AVALON_TIMER_CONTROL(REFRESH_TIMER_BASE,0x6);
//ENABLE REFRESH TIMER
        }

        if (packet[9] == 0x00)
        {
            printf(" \n");
        }
        else if (packet[9] >= 0xF0)
        {
            printf(" Fill \n");
            ctrlreg = packet[9] & 7;
            ctrlreg = ctrlreg*32;
            // CLEAR FILL_RGB BITS

IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0xE0);
            // SET FILL_RGB BITS

IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,ctrlreg);
        }
      }
      //else printf("Invalid header");
    //}
```

```
    packet[0] = 0;
  }
}


/*
 * static void TestLEDs(void)
 *
 * This function tests LED functionality.
 * It exits when the user types a 'q'.
 */

static void TestLEDs(void)
{
  volatile alt_u8 led;
  static char ch;
  static char entry[4];

  /* Turn the LEDs on. */
  led = 0x0;
  IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);
  //IOWR_ALTERA_AVALON_FIFO_DATA(EVAL_FIFO_MM_IN_BASE,0xDEADBEEF);
  //IOWR_ALTERA_AVALON_FIFO_DATA(EVAL_FIFO_MM_IN_BASE,0x55AA00FF);
  printf( "All LEDs should now be on.\r\n");
  printf( "Please press 'q' [Followed by <enter>] to exit this
test.\r\n");

  /* Get the input string for exiting this test. */
  do {
    GetInputString( entry, sizeof(entry), stdin);
    sscanf( entry, "%c\n", &ch );
  } while ( ch != 'q' );

  /* Turn the LEDs off and exit. */
  led = 0xff;
  IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led);
  printf(".....Exiting LED Test.\r\n");
}

/*****************************************
 * Button/Switch PIO Functions
 *****************************************/

/*****************************************************************
 * static void handle_button_interrupts( void* context, alt_u32 id)*
 *                                                                 *
 * Handle interrupts from the buttons.                             *
 * This interrupt event is triggered by a button/switch press.     *
 * This handler sets *context to the value read from the button    *
 * edge capture register.  The button edge capture register        *
 * is then cleared and normal program execution resumes.           *
 * The value stored in *context is used to control program flow    *
 * in the rest of this program's routines.                         *
 *****************************************************************/

static void handle_button_interrupts(void* context, alt_u32 id)
{
```

```
  /* Cast context to edge_capture's type.
   * It is important to keep this volatile,
   * to avoid compiler optimization issues.
   */
  int ctrlreg;

  volatile int* edge_capture_ptr = (volatile int*) context;
  /* Store the value in the Button's edge capture register in *context.
*/
  *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
  /* Reset the Button's edge capture register. */
  //printf("Just captured a switch interrupt #%i\n",
*edge_capture_ptr);
  switch_edge_capture = *edge_capture_ptr;
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);
  IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE); //An extra read
call to clear of delay through the bridge
  //usleep(100000); //sleep 0.25 seconds to prevent double-hit

}

/* Initialize the button_pio. */

static void init_button_pio()
{
  /* Recast the edge_capture pointer to match the alt_irq_register()
function
  * prototype. */
  void* edge_capture_ptr = (void*) &switch_edge_capture;
  /* Enable top 2 button interrupts. */
  IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
  /* Reset the edge capture register. */
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
  /* Register the interrupt handler. */
  alt_irq_register( BUTTON_PIO_IRQ, edge_capture_ptr,
handle_button_interrupts );
}

/* Tear down the button_pio. */

static void disable_button_pio()
{
  /* Disable interrupts from the button_pio PIO component. */
  IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0x0);
  /* Un-register the IRQ handler by passing a null handler. */
  alt_irq_register( BUTTON_PIO_IRQ, NULL, NULL );
}

/*****************************************************************
*********
 *
 * static void TestButtons( void )
 *
 * Generates a loop that exits when all buttons/switches have been
pressed,
 * at least, once.
 *
```

```
 * NOTE:  Buttons/Switches are not debounced.  A single press of a
 * button may result in multiple messages.
 *

**********************************************************************
*******/


static void TestButtons( void )
{
  alt_u8 buttons_tested;
  alt_u8 all_tested;
  /* Variable which holds the last value of edge_capture to avoid
   * "double counting" button/switch presses
   */
  int last_tested;
  /* Initialize the Buttons/Switches (SW0-SW3) */
  init_button_pio();
  /* Initialize the variables which keep track of which buttons have
been tested. */
  buttons_tested = 0x0;
  all_tested = 0xf;

  /* Initialize edge_capture to avoid any "false" triggers from
   * a previous run.
   */

  switch_edge_capture = 0;

  /* Set last_tested to a value that edge_capture can never equal
   * to avoid accidental equalities in the while() loop below.
   */

  last_tested = 0xffff;

  /* Print a quick message stating what is happening */

  printf("A loop will be run until all buttons/switches have been
pressed.                ");
  printf("NOTE:  Once a button press has been detected, for a
particular button,          ");
  printf("any further presses will be ignored!
");

  /* Loop until all buttons have been pressed.
   * This happens when buttons_tested == all_tested.
   */

  while (  buttons_tested != all_tested )
  {
    if (last_tested == switch_edge_capture)
    {
      continue;
    }
    else
    {
      last_tested = switch_edge_capture;
```

```
      switch (switch_edge_capture)
      {
        case 0x1:
          if (buttons_tested & 0x1)
          {
            continue;
          }
          else
          {
            printf("Button 1 (SW0) Pressed.");
            buttons_tested = buttons_tested | 0x1;
          }
          break;
        case 0x2:
          if (buttons_tested & 0x2)
          {
            continue;
          }
          else
          {
            printf("Button 2 (SW1) Pressed.”);
            buttons_tested = buttons_tested | 0x2;
          }
          break;
        case 0x4:
          if (buttons_tested & 0x4)
          {
            continue;
          }
          else
          {
            printf("Button 3 (SW2) Pressed.");
            buttons_tested = buttons_tested | 0x4;
          }
          break;
        case 0x8:
          if (buttons_tested & 0x8)
          {
            continue;
          }
          else
          {
            printf("Button 4 (SW3) Pressed.");
            buttons_tested = buttons_tested | 0x8;
          }
          break;
      }
    }
  }
  /* Disable the button pio. */
  disable_button_pio();

  printf("All Buttons (SW0-SW3) were pressed, at least, once.”);
  //usleep(2000000);
  return;
}
```

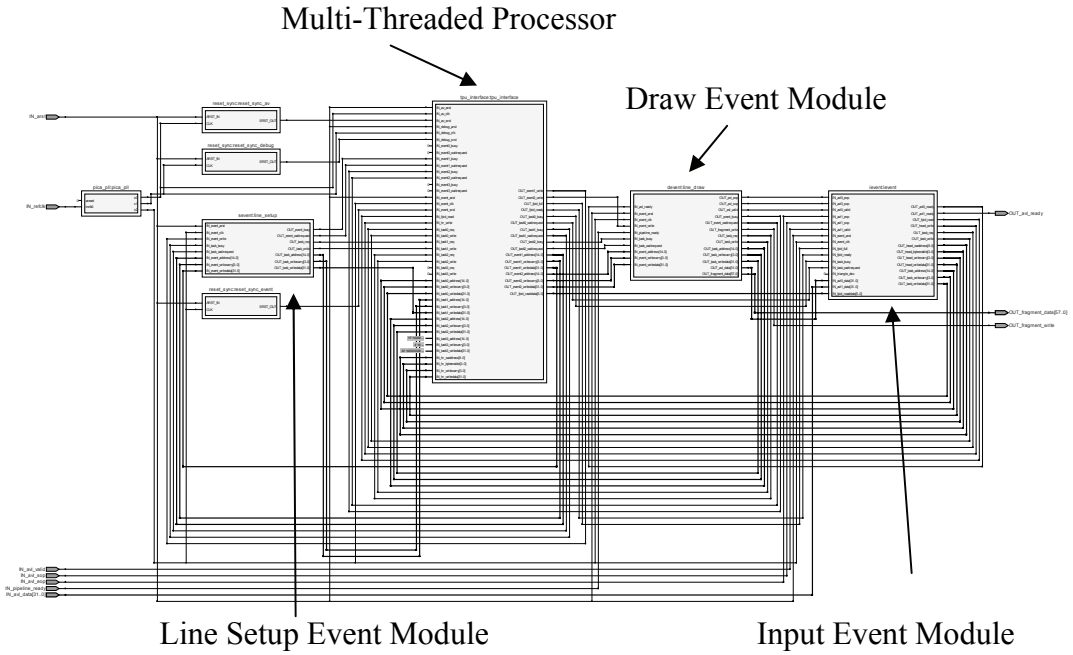
```
int main()
{

  IOWR_ALTERA_AVALON_TIMER_PERIODL(REFRESH_TIMER_BASE,0x25A0);
  IOWR_ALTERA_AVALON_TIMER_PERIODH(REFRESH_TIMER_BASE,0x0026);
  IOWR_ALTERA_AVALON_TIMER_CONTROL(REFRESH_TIMER_BASE,0x6);  //ENABLE
REFRESH TIMER

  IOWR_ALTERA_AVALON_TIMER_PERIODL(DIAGNOSTIC_TIMER_BASE,0xFFFF);
  IOWR_ALTERA_AVALON_TIMER_PERIODH(DIAGNOSTIC_TIMER_BASE,0xFFFF);

  //fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);

  //bmp_port = fopen("/dev/uart_0", "w");
  int uart_baud = 651;
  //int uart_baud = 217;
  IOWR_ALTERA_AVALON_UART_DIVISOR(UART_0_BASE,uart_baud);


  display_list[0][0] = 0;
  display_list[1][0] = 0;
  display_list[2][0] = 0;
  display_list[3][0] = 0;
  display_list[4][0] = 0;
  display_list[5][0] = 0;
  display_list[6][0] = 0;
  display_list[7][0] = 0;
  display_list_call[0] = 0;
  display_list_call[1] = 0;
  display_list_call[2] = 0;
  display_list_call[3] = 0;
  display_list_call[4] = 0;
  display_list_call[5] = 0;
  display_list_call[6] = 0;
  display_list_call[7] = 0;

  //binary_mode = 1;

  init_button_pio();
  init_ctrl_pio();

  while (1)
  {
    if (binary_mode)
    {
        ProcessBinary();
    }
    else
    {
        ProcessMenus();
    }
    if ((bmp_mode)&&(received_bmp_data))
    {
        bitmap_pixel = IORD_ALTERA_AVALON_PIO_DATA(BITMAP_BUS_BASE);
        //fprintf(bmp_port,"%u\n",bitmap_pixel);
```

```
        printf("%u\n",bitmap_pixel);
        //fflush(stdout);
        usleep(500);
        //if (ferror(bmp_port)) printf("Error outputing to bmp_port");

        // PULSE GET_PIXEL HIGH
        IOWR_ALTERA_AVALON_PIO_SET_BITS(GP_CTRL_OUTPUTS_BASE,0x200);
        IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(GP_CTRL_OUTPUTS_BASE,0x200);

        received_bmp_data = 0;
//            usleep(100000);
        //printf("%i: %u\n",pixel_counter,bitmap_pixel);
    }

    if (capture_finish_timer)
    {
        printf("finished timer = %.0f", elapsed_time);
        capture_finish_timer = 0;
    }

  }
  return( 0 );
}
```

Appendix D – Rasterizer Diagrams

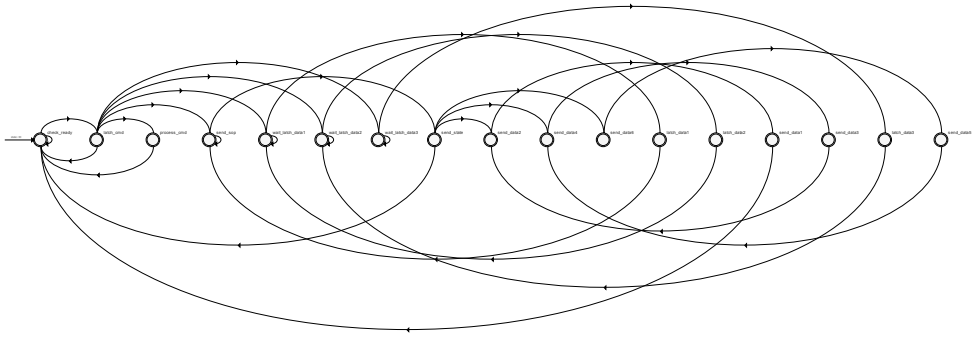
The Rasterizer's block diagrams and state machines shown below were created by the RTL Viewer in Altera's Quartus2 development environment.

Overall Rasterizer Block Diagram:

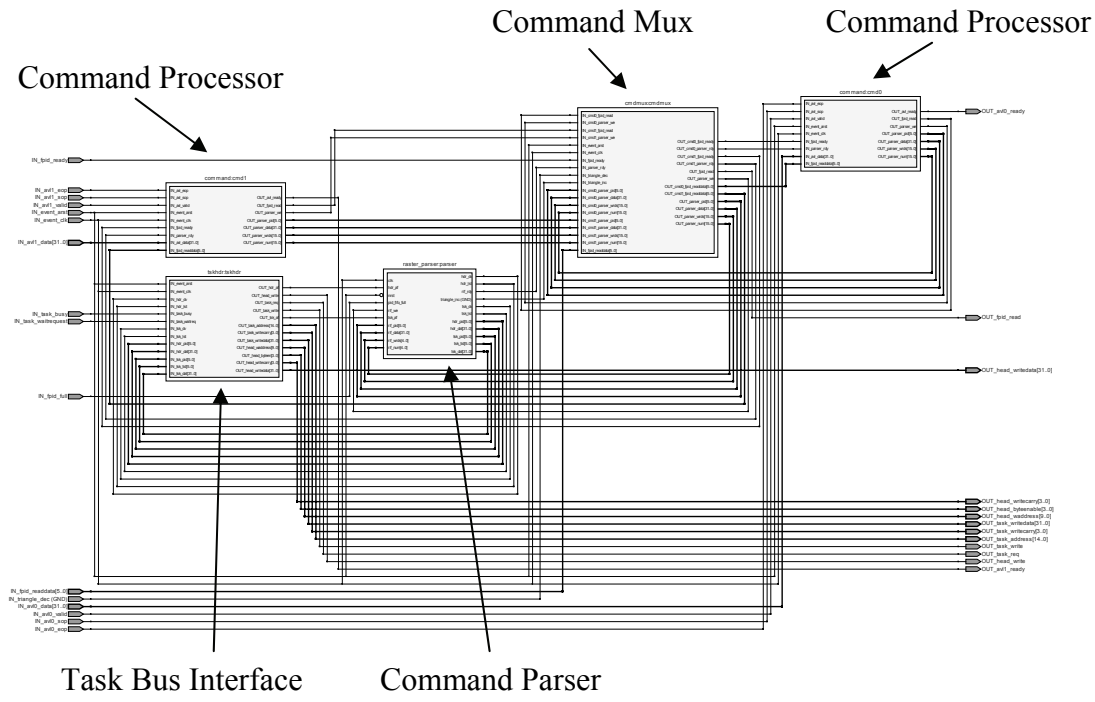


Rasterizer Pre-Processor State Machine:

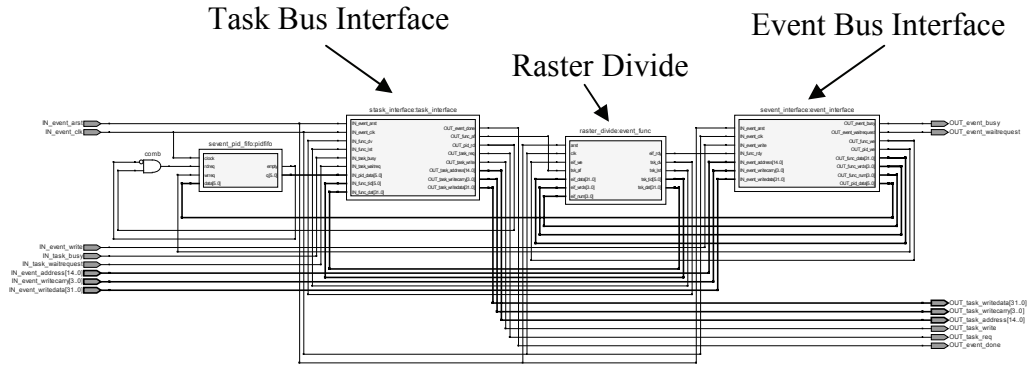
This state machine, which is located in the hierarchy at the same level as the Rasterizer, latches incoming command data from the FIFO and processes it accordingly. The support for multiple commands contributes to the multiple paths shown below in the state machine.



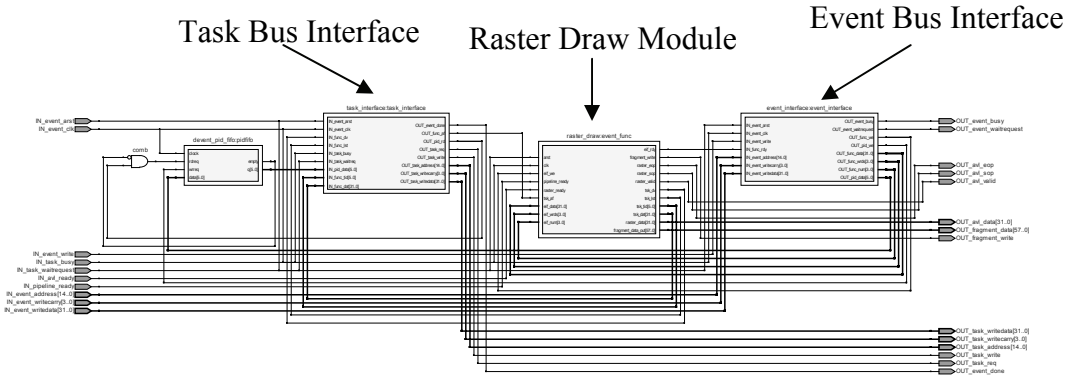
Rasterizer Input Event Module:



Setup Event Module:

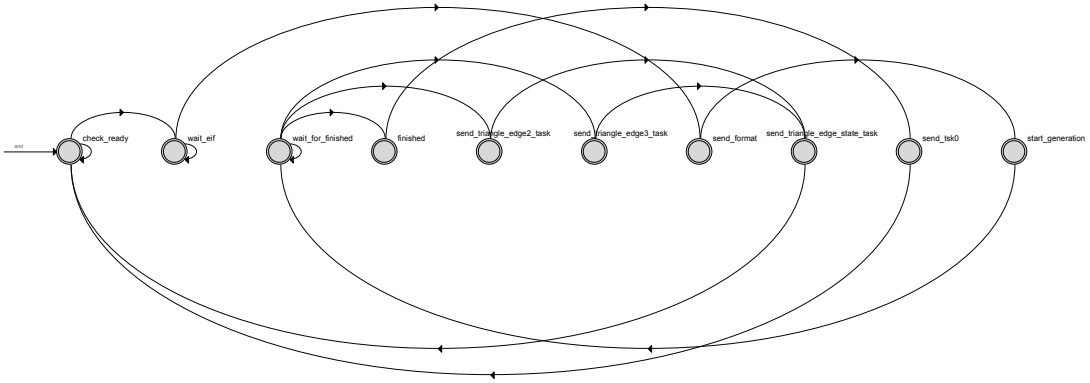


Draw Event Module:



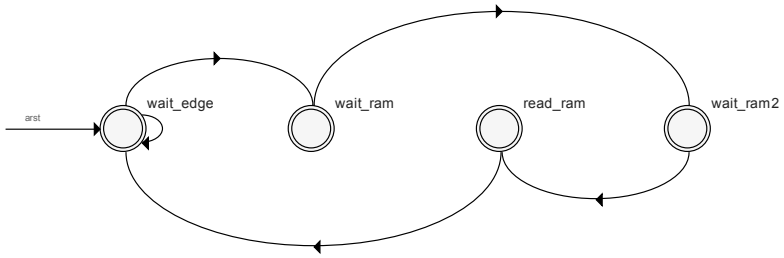
Raster Draw Module – Main State Machine:

This state machine waits for incoming events and then captures the appropriate data to send to the Line Generator.



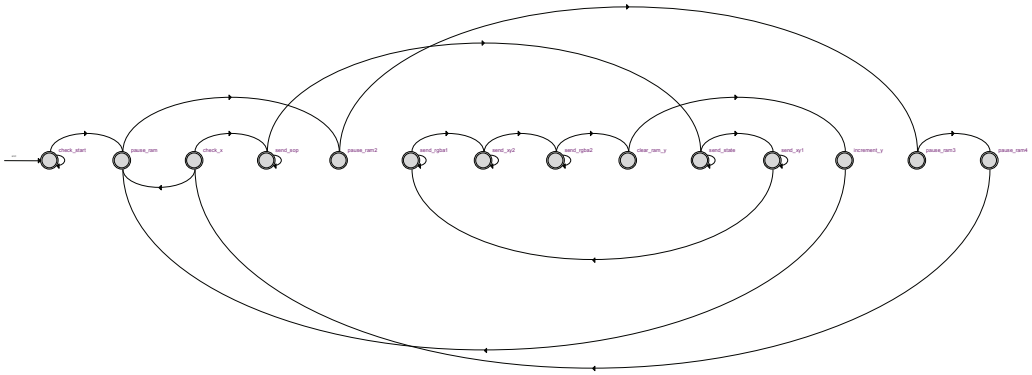
Raster Draw Module – Edges State Machine:

This state machine waits for edge pixels from the Line Generator and stores them in the Endpoint RAM if necessary.



Raster Draw Module – Fill State Machine:

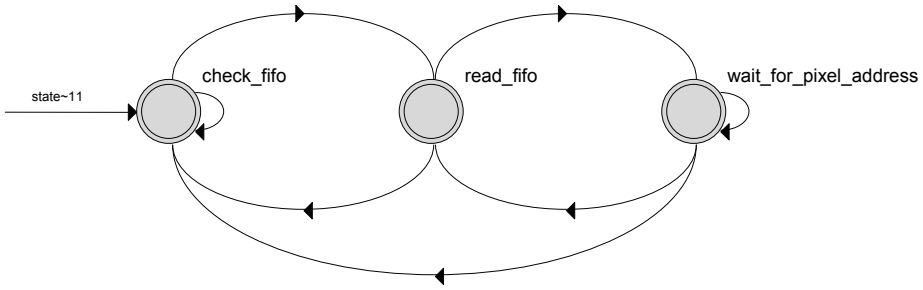
This state machine waits for the completion of all three triangle edges from the Line Generator. When it detects that condition, it will read the entire Endpoint RAM to determine where horizontal line fills are necessary. This state machine generates new line generation packets and sends them back to the Input Event Module.



Appendix E – Frame Buffer Operator Diagrams

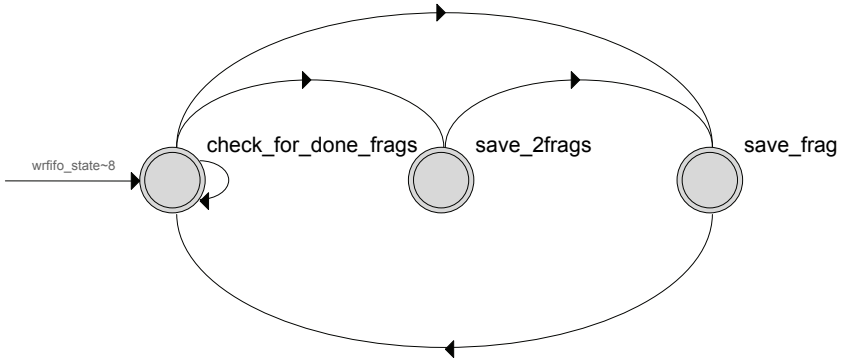
The Frame Buffer Operator's block diagrams and state machines shown below were created by the RTL Viewer in Altera's Quartus2 development environment. Due to the large amount of combinatorial logic in the Frame Buffer Operator, the top-level block diagram spans across five pages and is not suitable for inclusion in this appendix.

Main State Machine:



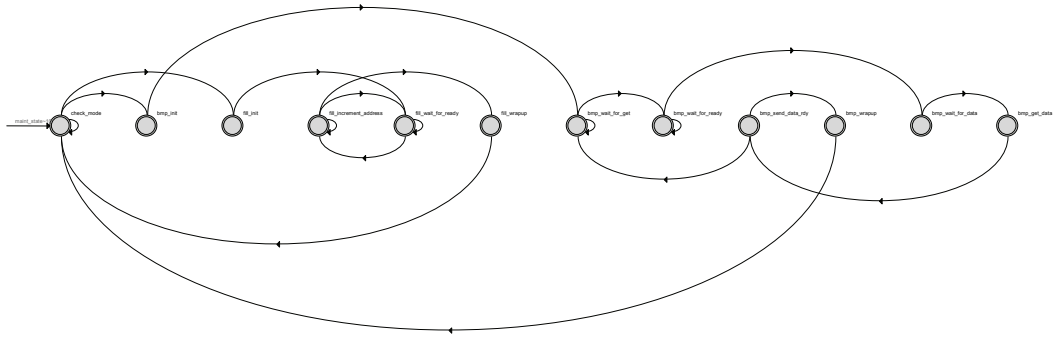
Write FIFO State Machine:

This state machine waits for fragments that have been processed by the Alpha Blender Manager. When a completed fragment is received, this state machine writes it to the Write Request FIFO.



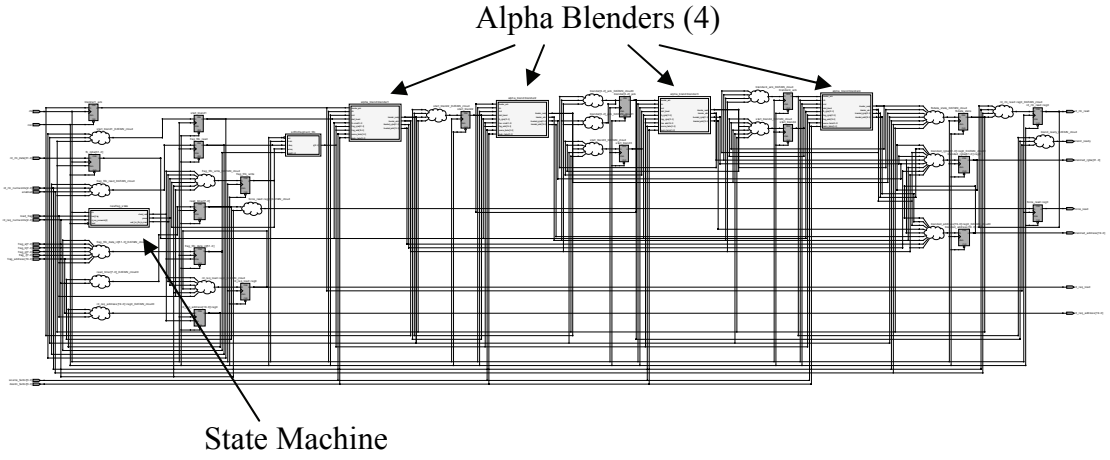
Diagnostic State Machine:

This state machine handles the bitmap capture process to allow the entire frame buffer contents to be transmitted out a serial port, one pixel at a time.



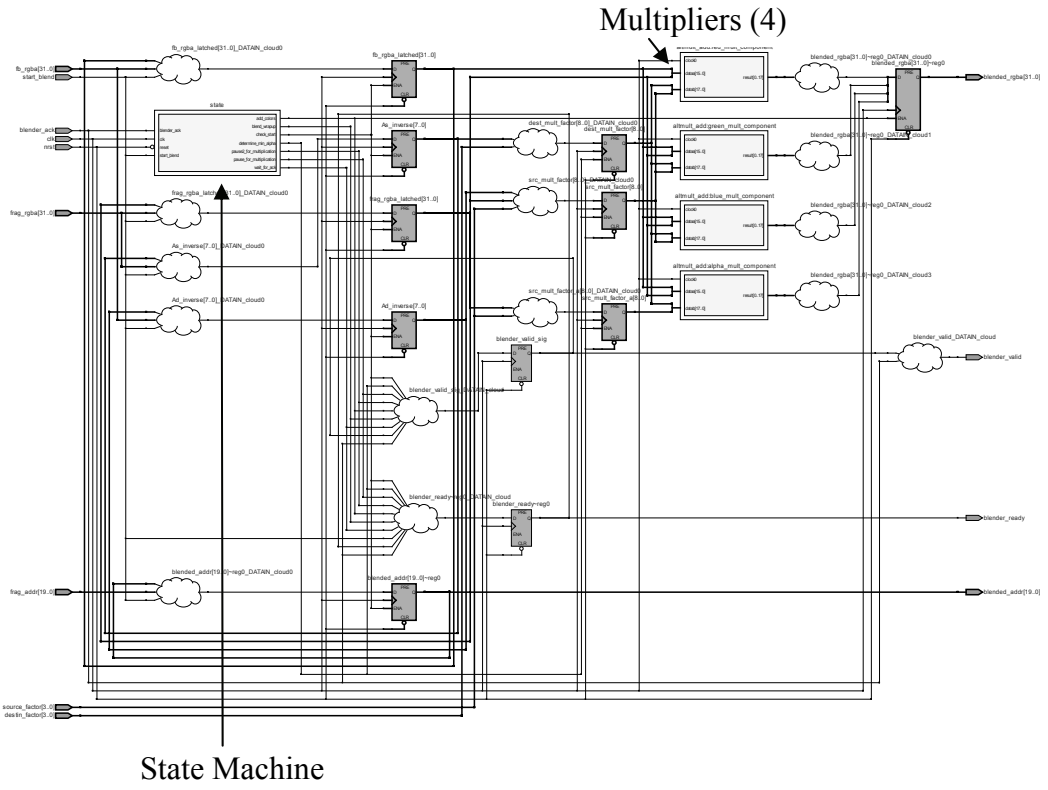
Alpha Blender Manager Block Diagram:

The Alpha Blender Manager is one of the components in the Frame Buffer Operator, and it has four Alpha Blender modules located in it.



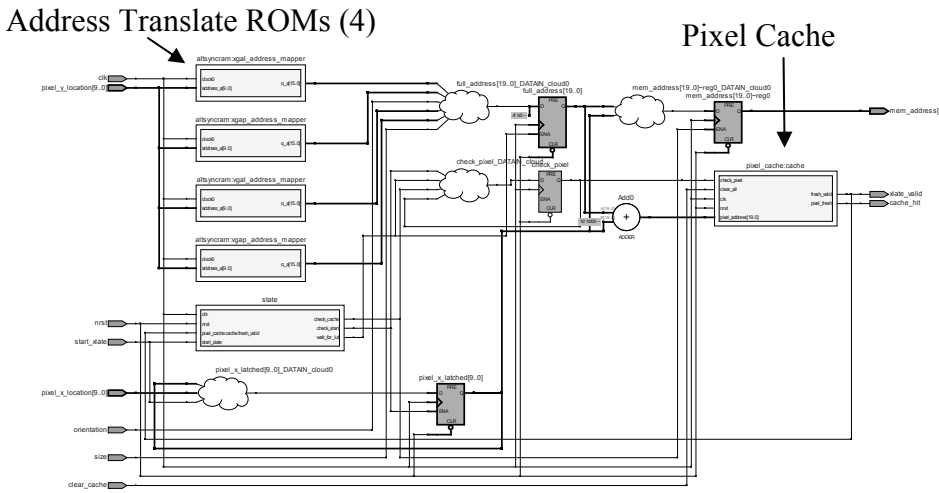
Alpha Blender Block Diagram:

The Alpha Blender is a component in the Alpha Blender Manager and actually performs the alpha blending operations.



Pixel Address Translate & Pixel Cache Block Diagram:

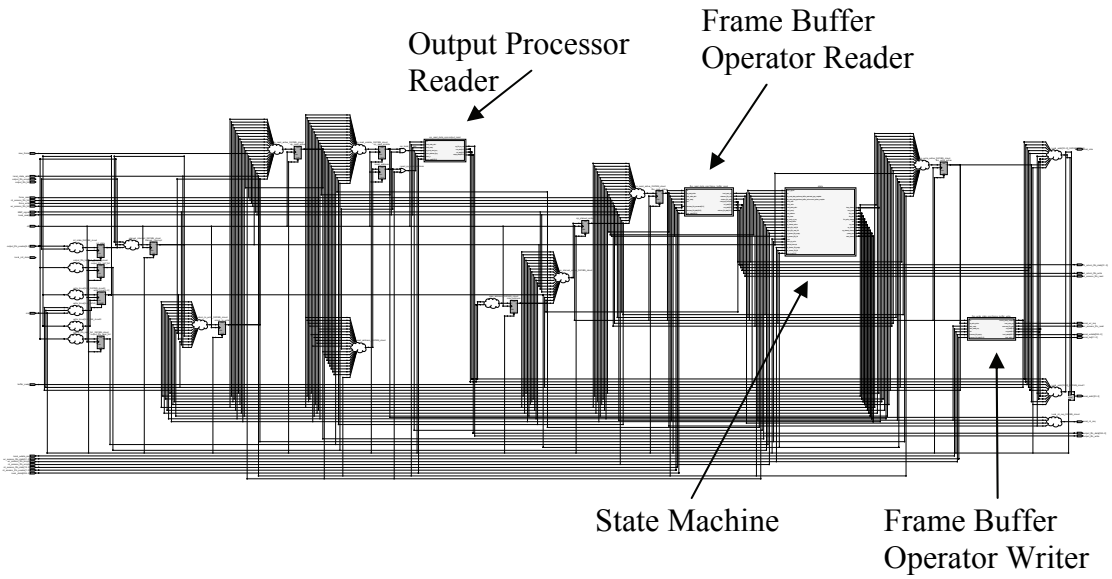
The Pixel Address Translate module is a component in the Frame Buffer Operator.



Appendix F – Memory Arbiter Diagrams

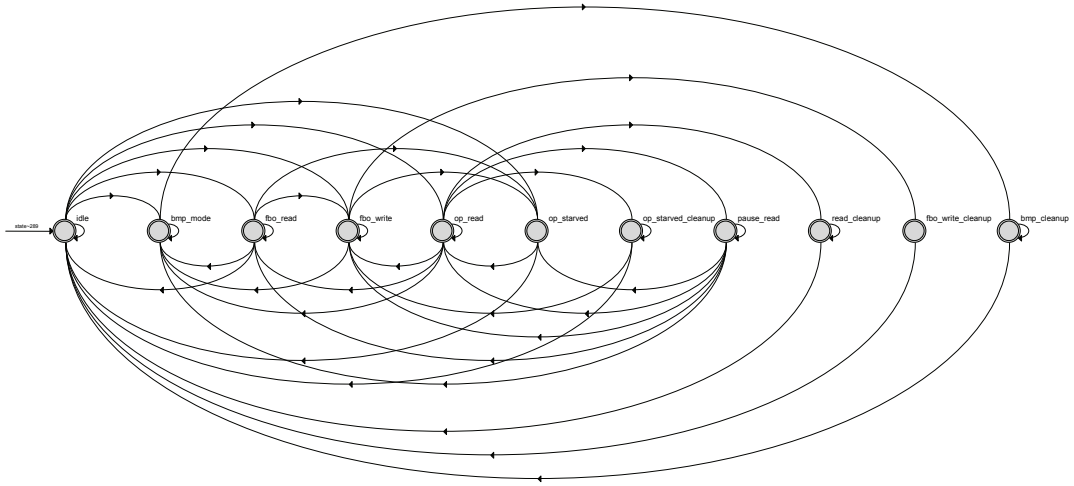
The Memory Arbiter's block diagram and state machine shown below were created by the RTL Viewer in Altera's Quartus2 development environment.

Overall Memory Arbiter Block Diagram:



Overall State Machine:

This state machine coordinates priorities between the Frame Buffer Operator and Output Processor, depending on which operation is underway and the starvation level of the Output Processor FIFO.

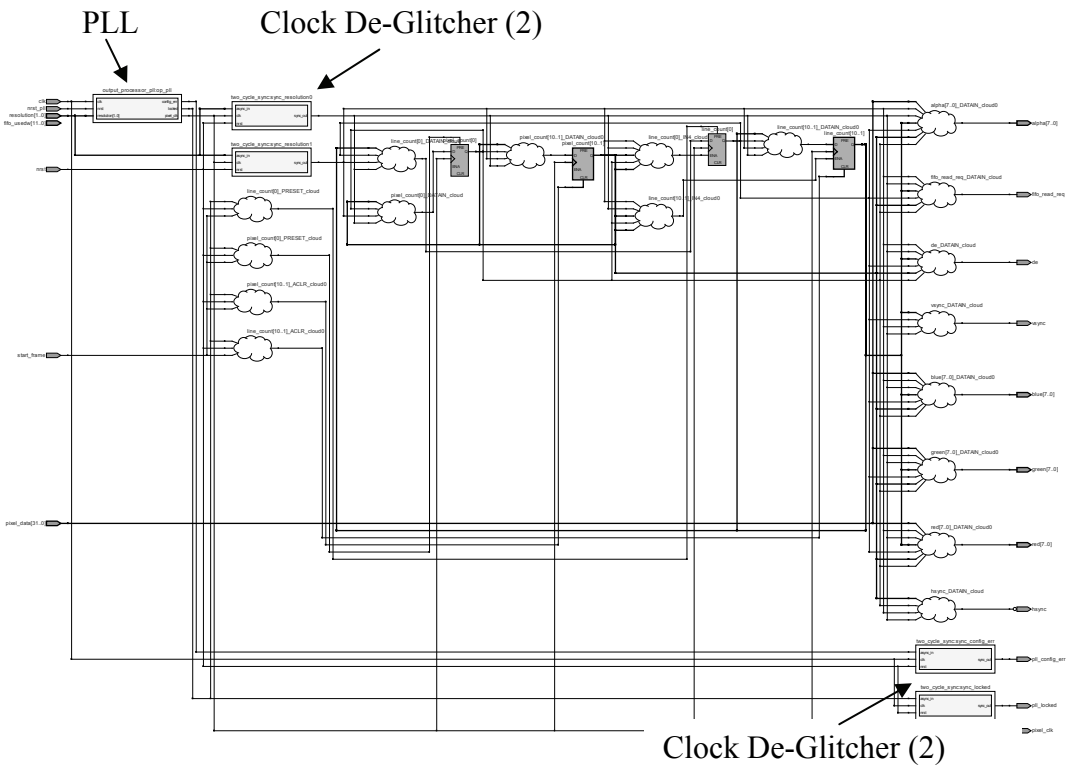


Appendix G – Output Processor Diagram

The Output Processor's block diagram shown below was created by the RTL Viewer in Altera's Quartus2 development environment.

Overall Block Diagram:

The Output Processor's RTL block diagram shows mostly combinatorial logic along with a PLL and several two-cycle synchronization de-glitching modules to cross clock boundaries appropriately.



Appendix H - TXT2BMP Utility Source Code

The following code was used to create the TXT2BMP command-line utility software used during the automated simulation approach described in Chapter 4.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <bitset>

using namespace std;

ifstream::pos_type size;
char * memblock;
char * bitmap;
int raster_format, bitmap_format, glint_format, landscape;
ifstream bmpfile;
int xga_size = 0;
int c = 0;
int tempascii = 0;

int main(int argc, char *argv[])
{
   if (argc < 7)
   {
      cout <<"TXT2BMP Syntax: <input_filename.dat>
<output_filename.bmp> <-r|f|b|g> <-x|v> <-b|w> <-l|p>\n";
   }
   else
   {
      if (!(strcmp(argv[3],"-r")))
      {
         cout << "Raster format";   //rasterizer format is basically
         //the xy coordinates in first 4 bytes of pixel data
         glint_format = 0;
         raster_format = 1;
         bitmap_format = 0;
      }
      else if (!(strcmp(argv[3],"-f")))
      {
         cout << "Frame Buffer format";   //fb format is the pixel
         //address in memory in first 3 bytes of pixel data
         glint_format = 0;
         raster_format = 0;
         bitmap_format = 0;
      }
      else if (!(strcmp(argv[3],"-b")))
      {
         cout << "BMP capture format";    //bmp format is all pixel
         //data in sequential order with commas
         glint_format = 0;
```

```
   raster_format = 0;
   bitmap_format = 1;
}
else if (!(strcmp(argv[3],"-g")))
{
   cout << "GLINT Rasterizer format";     //bmp format is all
   //pixel data in sequential order with commas
   glint_format = 1;
   raster_format = 0;
   bitmap_format = 0;
}
else {
   cout <<"TXT2BMP Syntax: <input_filename.dat>
   <output_filename.bmp> <-r|f> <-x|v> <-b|w> <-l|p>\n";
   return 0;
}

if (!(strcmp(argv[6],"-l")))
{
   landscape = 1;
}
else if (!(strcmp(argv[6],"-p")))
{
   landscape = 0;
}
else {
   cout <<"TXT2BMP Syntax: <input_filename.dat>
   <output_filename.bmp> <-r|f> <-x|v> <-b|w> <-l|p>\n";
   return 0;
}

if (!(strcmp(argv[4],"-x")))
{
   xga_size = 1;
   if (!(strcmp(argv[5],"-b")))
   {
      if (landscape)
      {
         cout << " with black xga in landscape.\n";
         bmpfile.open("xga_black_land.bmp",
         ios::in|ios::binary|ios::ate);
      }
      else
      {
         cout << " with black xga in portrait.\n";
         bmpfile.open("xga_black_port.bmp",
         ios::in|ios::binary|ios::ate);
      }
   }
   else if (!(strcmp(argv[5],"-w")))
   {
      if (landscape)
      {
         cout << " with white xga in landscape.\n";
         bmpfile.open("xga_white_land.bmp",
         ios::in|ios::binary|ios::ate);
      }
```

```
      else
      {
         cout << " with white xga in portrait.\n";
         bmpfile.open("xga_white_port.bmp",
         ios::in|ios::binary|ios::ate);
      }
   }
   else
   {
      cout <<"TXT2BMP Syntax: <input_filename.dat>
      <output_filename.bmp> <-r|f> <-x|v> <-b|w> <-l|p>\n";
      return 0;
   }
}
else if (!(strcmp(argv[4],"-v")))
{
   xga_size = 0;
   if (!(strcmp(argv[5],"-b")))
   {
      if (landscape)
      {
         cout << " with black vga bitmap in landscape.\n";
         bmpfile.open("vga_black_land.bmp",
         ios::in|ios::binary|ios::ate);
      }
      else
      {
         cout << " with black vga bitmap in portrait.\n";
         bmpfile.open("vga_black_port.bmp",
         ios::in|ios::binary|ios::ate);
      }
   }
   else if (!(strcmp(argv[5],"-w")))
   {
      if (landscape)
      {
         cout << " with white vga bitmap in landscape.\n";
         bmpfile.open("vga_white_land.bmp",
         ios::in|ios::binary|ios::ate);
      }
      else
      {
         cout << " with white vga bitmap in portrait.\n";
         bmpfile.open("vga_white_port.bmp",
         ios::in|ios::binary|ios::ate);
      }
   }
   else
   {
      cout <<"TXT2BMP Syntax: <input_filename.dat>
      <output_filename.bmp> <-r|f> <-x|v> <-b|w> <l|p>\n";
      return 0;
   }
}
else
{
```

```
            cout <<"TXT2BMP Syntax: <input_filename.dat>
            <output_filename.bmp> <-r|f> <-x|v> <-b|w> <l|p>\n";
            return 0;
         }

         ifstream infile (argv[1], ios::in|ios::binary|ios::ate);
         ofstream outfile (argv[2], ios::out|ios::binary);
         if (infile.is_open())
         {
            size = infile.tellg();
            memblock = new char [size];
            infile.seekg (0, ios::beg);
            infile.read (memblock, size);
            infile.close();
            cout << "Read input text file with " << size << " bytes ";
            int num_pixels = size/66;
            //int num_pixels = 17;
            cout << "and therefore " << num_pixels << " pixels.\n";

            if (bmpfile.is_open())
            {
               size = bmpfile.tellg();
               bitmap = new char [size];
               bmpfile.seekg(0, ios::beg);
               bmpfile.read(bitmap, size);
               bmpfile.close();

               cout << "Read starting bitmap file of size " << size <<
".\n\n";
            }
            else
            {
               cout << "Unable to open bitmap file.\n";
               return 0;
            }

            infile.open(argv[1], ios::in);
            infile.seekg(0, ios::beg);
            int pixel_ctr = 0;
            int i;
            int xf, yf, pf, af;
            int pix_offset, shift;
            bitset<16> x, y;
            bitset<8> x2, y2;
            bitset<24> pixaddress;
            xf = 0;
            if ((landscape) && (xga_size)) yf = 767;
            if (!(landscape) && (xga_size)) yf = 1023;
            if ((landscape) && !(xga_size)) yf = 479;
            if (!(landscape) && !(xga_size)) yf = 639;

            char pix_chars[11];
            double pix_rgbaf;
            int num_lines = 0;
            string line;

            if (glint_format)
```

```
{
   while (! infile.eof() )
   {
      getline (infile, line);
      stringstream myStream(line);
      myStream>>xf;
      getline (infile, line);
      stringstream myStreamy(line);
      myStreamy>>yf;
      getline (infile, line);
      stringstream myStreama(line);
      myStreama>>af;

      if (xga_size)
      {
         if (landscape)
         {
            if ((yf>767)||(yf<0)||(xf>1023)||(xf<0))
            {
               cout << "\n*** X or Y is out of bounds ***\n";
               return 0;
            }
            yf = 767 - yf;
            pix_offset = 54 + 3*(xf + (yf*1024));
         }
         else
         {
            if ((xf>1023)||(xf<0)||(yf>767)||(yf<0))
            {
               cout << "\n*** X or Y is out of bounds ***\n";
               return 0;
            }
            yf = 1023 - yf;
            pix_offset = 54 + 3*(xf + (yf*768));
         }
      }
      else
      {
         if (landscape)
         {
            if ((yf>479)||(yf<0)||(xf>639)||(xf<0))
            {
               cout << "\n*** X or Y is out of bounds ***\n";
               return 0;
            }
            yf = 479 - yf;
            pix_offset = 54 + 3*(xf + (yf*640));
         }
         else
         {
            if ((xf>639)||(xf<0)||(yf>479)||(yf<0))
            {
               cout << "\n*** X or Y is out of bounds ***\n";
               return 0;
            }
            yf = 639 - yf;
            pix_offset = 54 + 3*(xf + (yf*480));
```

```
         }
      }

      bitmap[pix_offset] = af;
      bitmap[pix_offset+1] = af;
      bitmap[pix_offset+2] = af;

   }
}
else if (bitmap_format)
{

   while (infile.getline(pix_chars,11))
   {
      num_lines++;
      cout << "pix_chars = " << pix_chars;
      pix_rgbaf = atof(pix_chars);
      //cout << "  pix_rgba = " << pix_rgbaf;
      double pix_rgbf = pix_rgbaf/256;
      double pix_rg = pix_rgbf/256;
      int pix_r = pix_rg/256;
      //cout << " R: " << pix_r;
      int pix_g = pix_rg - (pix_r*256);
      //cout << " G: " << pix_g;
      int pix_b = pix_rgbf - (pix_g*256) - (pix_r*256*256);
      //cout << " B: " << pix_b;

      if ((landscape) && (xga_size))
      {
         pix_offset = 54 + 3*(xf + (yf*1024));
         if (xf < 1023)
         {
            xf++;
         }
         else
         {
            xf = 0;
            yf--;
         }
      }
      if (!(landscape) && (xga_size))
      {
         pix_offset = 54 + 3*(xf + (yf*768));
         if (xf < 767)
         {
            xf++;
         }
         else
         {
            xf = 0;
            yf--;
         }
      }
      if ((landscape) && !(xga_size))
      {
         pix_offset = 54 + 3*(xf + (yf*640));
         if (xf < 639)
```

```
            {
               xf++;
            }
            else
            {
               xf = 0;
               yf--;
            }
         }
         if (!(landscape) && !(xga_size))
         {
            pix_offset = 54 + 3*(xf + (yf*480));
            if (xf < 479)
            {
               xf++;
            }
            else
            {
               xf = 0;
               yf--;
            }
         }
         cout << " 0: " << pix_offset << "\n";

         bitmap[pix_offset] = pix_b;
         bitmap[pix_offset+1] = pix_g;
         bitmap[pix_offset+2] = pix_r;
      }
      cout << "number of lines processed: " << num_lines;
   }
   else
   {
      while (pixel_ctr < num_pixels)
      {
         shift = pixel_ctr*66;
         if (raster_format)
         {
            x.reset();
            y.reset();
            x2.reset();
            y2.reset();
            c = 0;
            while (c<16)
            {
               bitset<8> temp (memblock[shift+c]);
               tempascii = temp.to_ulong();
               //cout << " temp = " << temp.to_ulong() << endl;

               switch (tempascii)
               {
               case 48:
                  x.set(15-c,0);
                  break;
               case 49:
                  x.set(15-c,1);
                  break;
               default:
```

```
      cout << "found something other than 0 or 1";
      break;
   }
   c++;
}
cout << " X: " << x.to_ulong();

c = 0;
while (c<16)
{
   bitset<8> temp (memblock[shift+c+16]);
   tempascii = temp.to_ulong();
   //cout << " temp = " << temp.to_ulong() << endl;

   switch (tempascii)
   {
   case 48:
      y.set(15-c,0);
      break;
   case 49:
      y.set(15-c,1);
      break;
   default:
      cout << "found something other than 0 or 1";
      break;
   }
   c++;
}


cout << " Y: " << y.to_ulong();

xf = x.to_ulong();
yf = y.to_ulong();
if (xga_size)
{
   if (landscape)
   {
      if ((yf>767)||(yf<0)||(xf>1023)||(xf<0))
      {
         cout << "\n X or Y is out of bounds ***\n";
         return 0;
      }
      yf = 767 - yf;
      pix_offset = 54 + 3*(xf + (yf*1024));
   }
   else
   {
      if ((xf>1023)||(xf<0)||(yf>767)||(yf<0))
      {
         cout << "\n X or Y is out of bounds ***\n";
         return 0;
      }
      yf = 1023 - yf;
      pix_offset = 54 + 3*(xf + (yf*768));
   }
}
```

```
else
{
   if (landscape)
   {
      if ((yf>479)||(yf<0)||(xf>639)||(xf<0))
      {
         cout << "\n X or Y is out of bounds ***\n";
         return 0;
      }
      yf = 479 - yf;
      pix_offset = 54 + 3*(xf + (yf*640));
   }
   else
   {
      if ((xf>639)||(xf<0)||(yf>479)||(yf<0))
      {
         cout << "\n X or Y is out of bounds ***\n";
         return 0;
      }
      yf = 639 - yf;
      pix_offset = 54 + 3*(xf + (yf*480));
   }
}

bitset<8> red (memblock[shift+4]);
bitset<8> green (memblock[shift+5]);
bitset<8> blue (memblock[shift+6]);
bitset<8> alpha (memblock[shift+7]);

bitset<24> pixaddress;

c = 0;
tempascii = 0;

while (c<8)
{
   bitset<8> temp (memblock[shift+c+32]); //32 is the
   //red offset from pixel begin
   tempascii = temp.to_ulong();
   //cout << "temp = " << temp.to_ulong() << endl;

   switch (tempascii)
   {
   case 48:
      red.set(7-c,0);
      break;
   case 49:
      red.set(7-c,1);
      break;
   default:
      cout << "found something other than 0 or 1";
      break;
   }
   c++;
}

c = 0;
```

```
tempascii = 0;

while (c<8)
{
   bitset<8> temp (memblock[shift+c+40]); //40 is the
   green offset from pixel begin
   tempascii = temp.to_ulong();
   //cout << "temp = " << temp.to_ulong() << endl;

   switch (tempascii)
   {
   case 48:
      green.set(7-c,0);
      break;
   case 49:
      green.set(7-c,1);
      break;
   default:
      cout << "found something other than 0 or 1";
      break;
   }
   c++;
}

c = 0;
tempascii = 0;

while (c<8)
{
   bitset<8> temp (memblock[shift+c+48]); //48 is the
   blue offset from pixel begin
   tempascii = temp.to_ulong();
   //cout << "temp = " << temp.to_ulong() << endl;

   switch (tempascii)
   {
   case 48:
      blue.set(7-c,0);
      break;
   case 49:
      blue.set(7-c,1);
      break;
   default:
      cout << "found something other than 0 or 1";
      break;
   }
   c++;
}

c = 0;
tempascii = 0;

while (c<8)
{
   bitset<8> temp (memblock[shift+c+56]); //56 is the
   //alpha offset from pixel begin
   tempascii = temp.to_ulong();
```

```
            //cout << "temp = " << temp.to_ulong() << endl;

            switch (tempascii)
            {
            case 48:
               alpha.set(7-c,0);
               break;
            case 49:
               alpha.set(7-c,1);
               break;
            default:
               cout << "found something other than 0 or 1";
               break;
            }
            c++;
         }

         cout << " RED: " << red.to_ulong();
         cout << " GREEN: " << green.to_ulong();
         cout << " BLUE: " << blue.to_ulong();
         cout << " ALPHA: " << alpha.to_ulong();
         cout << " PIX OFFSET: " << pix_offset << ".\n";

         int blended_blue = (255-alpha.to_ulong()) +
((blue.to_ulong())*(alpha.to_ulong())/255);
         int blended_green = (255-alpha.to_ulong()) +
((green.to_ulong())*(alpha.to_ulong())/255);
         int blended_red = (255-alpha.to_ulong()) +
((red.to_ulong())*(alpha.to_ulong())/255);

         bitmap[pix_offset] = blended_blue;
         bitmap[pix_offset+1] = blended_green;
         bitmap[pix_offset+2] = blended_red;

      }
      else //must be frame buffer format
      {
         //cout << "memblock shift0" << memblock[shift+0];
         c = 0;
         tempascii = 0;
         shift = pixel_ctr*66;

         while (c<24)
         {
            bitset<8> temp (memblock[shift+c]);
            tempascii = temp.to_ulong();
            //cout << "temp = " << temp.to_ulong() << endl;

            switch (tempascii)
            {
            case 48:
               pixaddress.set(23-c,0);
               break;
            case 49:
               pixaddress.set(23-c,1);
               break;
            default:
```

```
      cout << "found something other than 0 or 1";
      break;
   }
   c++;
}

pf = pixaddress.to_ulong();
cout << "ADDR = " << pf << "  ";

if (xga_size)
{
   if (landscape)
   {
      yf = pf/1024;
      xf = pf - (yf*1024);
      cout << "X = " << xf << "  ";
      cout << "Y = " << yf;
      if ((yf>767)||(yf<0)||(xf>1023)||(xf<0))
      {
         cout << "\n X or Y is out of bounds ***\n";
         return 0;
      }
      yf = 767 - yf;
      pix_offset = 54 + 3*(xf + (yf*1024));
   }
   else
   {
      yf = pf/768;
      xf = pf - (yf*768);
      cout << "X = " << xf << "  ";
      cout << "Y = " << yf;
      if ((yf>1023)||(yf<0)||(xf>767)||(xf<0))
      {
         cout << "\n X or Y is out of bounds ***\n";
         return 0;
      }
      yf = 1023 - yf;
      pix_offset = 54 + 3*(xf + (yf*768));
   }
}
else
{
   if (landscape)
   {
      yf = pf/640;
      xf = pf - (yf*640);
      cout << "X = " << xf << "  ";
      cout << "Y = " << yf;
      if ((yf>479)||(yf<0)||(xf>639)||(xf<0))
      {
         cout << "\n X or Y is out of bounds ***\n";
         return 0;
      }
      yf = 479 - yf;
      pix_offset = 54 + 3*(xf + (yf*640));
   }
   else
```

```
   {
      yf = pf/480;
      xf = pf - (yf*480);
      cout << "X = " << xf << "  ";
      cout << "Y = " << yf;
      if ((yf>639)||(yf<0)||(xf>479)||(xf<0))
      {
         cout << "\n X or Y is out of bounds ***\n";
         return 0;
      }
      yf = 639 - yf;
      pix_offset = 54 + 3*(xf + (yf*480));
   }
}

bitset<8> red (memblock[shift+4]);
bitset<8> green (memblock[shift+5]);
bitset<8> blue (memblock[shift+6]);

c = 0;
tempascii = 0;

while (c<8)
{
   bitset<8> temp (memblock[shift+c+32]); //32 is the
   //red offset from pixel begin
   tempascii = temp.to_ulong();
   //cout << "temp = " << temp.to_ulong() << endl;

   switch (tempascii)
   {
   case 48:
      red.set(7-c,0);
      break;
   case 49:
      red.set(7-c,1);
      break;
   default:
      cout << "found something other than 0 or 1";
      break;
   }
   c++;
}

c = 0;
tempascii = 0;

while (c<8)
{
   bitset<8> temp (memblock[shift+c+40]); //40 is the
   //green offset from pixel begin
   tempascii = temp.to_ulong();
   //cout << "temp = " << temp.to_ulong() << endl;

   switch (tempascii)
   {
   case 48:
```

```
                green.set(7-c,0);
                break;
            case 49:
                green.set(7-c,1);
                break;
            default:
                cout << "found something other than 0 or 1";
                break;
            }
            c++;
        }

        c = 0;
        tempascii = 0;

        while (c<8)
        {
            bitset<8> temp (memblock[shift+c+48]); //48 is the
            //blue offset from pixel begin
            tempascii = temp.to_ulong();
            //cout << "temp = " << temp.to_ulong() << endl;

            switch (tempascii)
            {
            case 48:
                blue.set(7-c,0);
                break;
            case 49:
                blue.set(7-c,1);
                break;
            default:
                cout << "found something other than 0 or 1";
                break;
            }
            c++;
        }

        cout << " RED: " << red.to_ulong();
        cout << " GREEN: " << green.to_ulong();
        cout << " BLUE: " << blue.to_ulong();
        cout << " PIX OFFSET: " << pix_offset << ".\n";

        bitmap[pix_offset] = blue.to_ulong();
        bitmap[pix_offset+1] = green.to_ulong();
        bitmap[pix_offset+2] = red.to_ulong();

    }

    pixel_ctr++;
  }
}


if (outfile.is_open())
{
  outfile.write(bitmap, size);
}
```

```
         else cout << "Error opening " << argv[2] << ".\n";
      }
      else cout << "Unable to open file: " << argv[1] << ".\n";
   }
   return 0;
}
```

References

[1] "Comparison of ATI graphics processing units," [Online]. Available: http://en.wikipedia.org/wiki/List_of_ATI_cards. [Accessed: February 12, 2009].

[2] "Comparison of NVIDIA graphics processing units," [Online]. Available: http://en.wikipedia.org/wiki/List_of_NVIDIA_Graphics_Processing_Units. [Accessed: February 12, 2009].

[3] J. Brown, "High Performance Processor Development for Consumer Electronics," presented at Symposium on VLSI Circuits, 2007.

[4] K. Fatahalian, M. Houston, "A Closer Look at GPUs," *Communications of the ACM*, pp. 50-57, October 2008.

[5] J. Andrews, N. Baker, "Xbox 360 System Architecture," *IEEE Micro*, pp. 25-37, March 2006.

[6] "Xilinx Market Share Under 50% For First Time in 7 Years," [Online]. Available: http://www.electronicsweekly.com/Articles/2010/08/05/49217/xilinx-market-share-under-50-for-first-time-in-7-years.htm. [Accessed: September 5, 2010].

[7] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, P. Cheung, "Reconfigurable Computing: Architectures and Design Methods," *IEE Proceedings of Computers and Digital Techniques*, pp. 193-207, March 2005.

[8] N. Telle, W. Luk, R. Chung, "Customising Hardware Designs for Elliptic Curve Cryptography," *Lecture Notes in Computer Science*, pp. 453-476, 2004.

[9] J. Beeckler, W. Gross, "FPGA Particle Graphics Hardware," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[10] "Stratix V Device Family Overview," Altera Corporation, Inc., July 2010.

[11] F. Garzia, C. Brunelli, D. Rossi, J. Nurmi, "Implementation of a floating-point matrix-vector multiplication on a reconfigurable architecture," *Proceedings of the 15th Reconfigurable Workshop*, pp. 1-6, 2008.

[12] "Stratix IV Device Family Overview," Altera Corporation, Inc., March 2010.

[13] "Stratix III Device Family Overview," Altera Corporation, Inc., March 2010.

[14] "Stratix II Device Family Data Sheet," Altera Corporation, Inc., May 2007.

[15] "Stratix Device Family Data Sheet," Altera Corporation, Inc., January 2006.

[16] S. Belkacemi, K. Benkrid, D. Crookes, A. Benkrid, "Design and Implementation of a High Performance Matrix Multiplier Core for Xilinx Virtex FPGAs," presented at IEEE International Workshop on Computer Architectures for Machine Perception, 2003.

[17] M. Birla, "FPGA Based Reconfigurable Platform for Complex Image Processing," *Proceedings of the IEEE International Conference on Electro/Information Technology*, pp. 204-209, 2006.

[18] C. Iakovidou, V. Vonikakis, I. Andreadis, "FPGA Implementation of a Real-Time Biologically Inspired Image Enhancement Algorithm," *Journal of Real-Time Image Processing*, pp. 269-287, 2008.

[19] S. McBader, P. Lee, "An FPGA Implementation of a Flexible, Parallel Image Processing Architectures Suitable for Embedded Vision Systems," *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2003.

[20] T. Kanamori, H. Amano, M. Arai, Y. Ajioka, "A High Speed License Plate Recognition System on an FPGA," *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications*, pp. 554-557, 2007.

[21] M. Sen, Y. Hemaraj, W. Plishker, R. Shekhar, S. Bhattacharyyam, "Model-based Mapping of Reconfigurable Image Registration on FPGA Platforms," *Journal of Real-Time Image Processing*, pp. 149-162, 2008.

[22] D. Blythe, "Rise of the Graphics Processor," *Proceedings of the IEEE*, Vol. 96, No. 5, pp. 761-778, May 2008.

[23] "Asteroids (video game) – Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Asteroids_(video_game). [Accessed: July 2, 2009].

[24] A. M. Noll, "Scanned-display computer graphics," *Communications of the ACM*, vol. 14, no. 3, pp. 143–150, March 1971.

[25] R. Pike, B. N. Locanthi, and J. Reiser, "Hardware/software trade-offs for bitmap graphics on the blit," *Software – Practice and Experience*, vol. 15, no. 2, pp. 131–151, 1985.

[26] "Pac-Man – Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Pac-Man. [Accessed: July 2, 2009].

[27] M. Segal and K. Akeley, "The OpenGL graphics system: A specification," Silicon Graphics, Inc., 1992–2006.

[28] Microsoft. (2006). Direct3D 10 Reference. [Online]. Available: http://www.msdn.microsoft.com/directx. [Accessed: March 20, 2010].

[29] "NVIDIA Technical Brief | NVIDIA nfiniteFX Engine: Programmable Pixel Shaders," [Online]. Available: http://www.nvidia.com/page/pg_20010529782175.html. [Accessed March 17, 2009].

[30] E. Lindholm, M. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," *Proceedings of ACM SIGGRAPH*, pp. 149–158, August 2001.

[31] F. Chehimi, P. Coulton, R. Edwards, "Evolution of 3D Mobile Games Development", *Personal and Ubiquitous Computing*, Vol. 12, pp. 19-25, January 2008.

[32] M. Segal, K. Akeley, "The Design of the OpenGL Graphics Interface," Silicon Graphics, Inc., 1994.

[33] J. Neider, M. Woo, and T. Davis, *OpenGL Programming Guide*. Addison-Wesley, Reading, MA, 1993.

[34] "Existing Fixed-function Pipeline Diagram," [Online]. Available: http://www.khronos.org/opengles/2_x/img/opengles_1x_pipeline.gif. [Accessed: October 17, 2008].

[35] M. Benvegnu, "Introduction to Shaders," presented at 5th Annual Summer School for Scientific Visualization and 3D Interactive Graphics, 2005.

[36] "ES2.0 Programmable Pipeline Diagram," [Online]. Available: http://www.khronos.org/opengles/2_x/img/opengles_20_pipeline.gif. [Accessed: October 17, 2008].

[37] "Vincent SC | Download Vincent SC software," [Online]. Available: http://sourceforge.net/projects/oglsc/. [Accessed July 7, 2010].

[38] L. Seiler, et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, August 2008.

[39] "Intel's Larrabee Chip is Dead, At Least as a Product," [Online]. Available: http://www.pcmag.com/article2/0,2817,2356725,00.asp. [Accessed: June 17, 2010].

[40] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, "Nvidia Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, pp. 39-55, March 2008.

[41] "NVIDIA's Next Generation CUDA Computer Architecture: Fermi," NVIDIA White Paper, Version 1.1. [Online]. Available: http://www.nvidia.com/ content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_ Architecture_Whitepaper. pdf. [Accessed August 1, 2010].

[42] T. Kim, K. Oh, "Design of a Programmable Vertex Processing Unit for Mobile Platforms." *International Federation for Information Processing*, pp. 805-814, 2006.

[43] T. Weyrich, et al., "A Hardware Architecture for Surface Splatting," *ACM Transactions on Graphics*, Vol. 26, No. 3, Article 90, July 2007.

[44] J. Eyles, et al., "PixelFlow: The Realization," *Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS Workshop on Graphics Hardware*, pp. 57-68, 1997.

[45] "GPGPU.org: General Purpose computation on Graphics Processing Units," [Online]. Available: http://gpgpu.org. [Accessed: February 25, 2009].

[46] J. Owens, "GPU Architecture Overview," presented at ACM SIGGRAPH, 2007.

[47] S. Stone, H. Yi, J. Haldar, W. Hwu, B. Sutton, Z. Liang, "Accelerating Advanced MRI Reconstructions on GPUs," *Proceedings of the 5th International Conference on Computing Frontiers*, May 2008.

[48] K. Mueller, Fang Xu, Neophytos Neophytou, "Why do Commodity Graphics Hardware Boards (GPUs) work so well for acceleration of Computed Tomography?" presented at SPIE Electronic Imaging Conference, 2007.

[49] T. Halfhill, "Parallel Processing with CUDA," *Microprocessor Report* [Available Online: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf], January 2008.

[50] A. Kerr, D. Campbell, M. Richards, "QR Decomposition on GPUs," *Proceedings of the 2nd ACM Workshop on General Purpose Processing on Graphics Processing Units,* pp. 71-78, 2009.

[51] "OpenCL – The Open Standard for Heterogeneous Parallel Programming," [Online]. Available: http://www.khronos.org/opencl/ [Accessed: April 1, 2009].

[52] R. Gu, T. Yeh, et. al., "A Low Cost Tile-based 3D Graphics Full Pipeline with Real-time Performance Monitoring Support for OpenGL ES in Consumer Electronics," *Proceedings of the IEEE International Symposium on Consumer Electronics*, pp. 1-6, 2007.

[53] "ARM Information Center," [Online]. Available: http://infocenter.arm.com/ help/index.jsp?topic=/com.arm.doc.ddi0068c/index.html. [Accessed May 27, 2010].

[54] C. Lee, E. Kim, "Design of a Geometry Engine for Mobile 3D graphics," *Proceedings of the International SOC Design Conference,* 2008.

[55] M. White, M. Waller, G. Dunnett, P. Lister, R. Grimsdale, "Graphics ASIC Design Using VHDL", *Computer & Graphics*, Elsevier Science Ltd., Vol. 19, pp. 301-308, 1995.

[56] S. Singh, P. Bellec, "Virtual Hardware for Graphics Applications Using FPGAs," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 49, 1994.

[57] H. Styles, W. Luk, "Customising Graphics Applications: Techniques and Programming Interfaces," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 77, 2000.

[58] D. Thomas, W. Luk, "Implementing Graphics Shaders Using FPGAs," *Lecture Notes in Computer Science*, Volume 3203, pp. 1173, 2004.

[59] N. Knutsson, "An FPGA-based 3D Graphics System," M.S. thesis, Linkoping Institute of Technology, Sweden, 2005.

[60] Y. Won, J. Park, W. Moon, "3-D Accelerator on Chip," Altera Nios II Embedded Processor Design Contest, pp. 109-117, 2005..

[61] L. Middendorf, F. Muhlbauer, G. Umlauf, C. Bobda, "Embedded Vertex Shader in FPGA," *Proceedings of International Federation for Information Processing*, Vol. 231, pp. 155-164, 2007.

[62] K. Owyang, "A High Speed Graphics Processing Application Using FPGAs," Actel Corporation, Inc., 1994.

[63] "D/AVE 2D FPGA Graphics Renderer," [Online]. Available: http://tes-dst.com/tes-dst/technology-a-ip/graphics-rendering/dave-2d.html. [Accessed March 3, 2010].

[64] "D/AVE 3D ASIC & FPGA Graphics IP," [Online]. Available: http://tes-dst.com/ tes-dst/technology-a-ip/graphics-rendering/dave-3d.html. [Accessed March 3, 2010].

[65] "IP Cores – logi3D," [Online]. Available: http://www.logicbricks.com/Products/ logi3D.aspx. [Accessed March 4, 2010].

[66] M. Dutton, D. Keezer, "An Architecture for Graphics Processing in an FPGA," *Proceedings of the 18th ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* 2010.

[67] M. Dutton, "System and Method for Flexible Architecture for Graphics Processing," U.S. Patent Pending, February 2011.

[68] M. Dutton, "A Scalable Architecture for Rasterization in an FPGA," U.S. Patent Pending, February 2011.

[69] J. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, Vol. 4, No. 1, pp. 25-30, January 1965.

[70] X. Wu, "An Efficient Antialiasing Technique," *Computer Graphics*, July 1991.

[71] "Midpoint circle algorithm," [Online]. Available: http://en.wikipedia.org/wiki/Midpoint_circle_algorithm. [Accessed January 3, 2009].

[72] "DDR and DDR2 SDRAM Controller Compiler User Guide v9.0," Altera Corporation, Inc., March 2009.

[73] "AN 454: Implementing PLL Reconfiguration in Stratix III and Stratix IV Devices," Altera Corporation, Inc., December 2009.

[74] M. Dutton, D. Keezer, "A Simulation Approach for Developing FPGA-Based Graphical & Image Processors," pending acceptance by *IEEE Computer Architecture Letters*, 2011.

[75] A. Zuloaga, J. Martin, U. Bidarte, A. Ezquerra, "VHDL test bench for digital image processing system using a new image format," University of the Basque Country, Department of Electronics and Telecommunications, 1998.

[76] "BMP file format – Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/BMP_file_format. [Accessed August 3, 2009].

[77] "Use of COTS Graphical Processor (CGP) in Airborne Display Systems," Certification Authorities Software Team, Position Paper CAST-29, February 2007.

[78] M. Dutton, D. Keezer, "The Challenges of Graphics Processing in the Avionics Industry," *Proceedings of the 29th IEEE/AIAA Digital Avionics Systems Conference*, pp. 5.A.1-1 – 5.A.1-9, 2010. [Awarded Best Paper of Session]

[79] K. Runyon, "Advanced Graphics Processor," *Proceedings of the 12th IEEE Digital Avionics Systems Conference*, 2003.

[80] "L-3 Display Systems," [Online]. Available: http://www.l-3com.com/displays/products/laad.htm. [Accessed: August 2, 2010].

[81] "SmartDeck Integrated Flight Controls & Display System," [Online]. Available: http://www.as.l-3com.com/downloads/brochures__pilots_guides/smartdeck_brochure.pdf. [Accessed: July 30, 2010].

[82] “DO-178B – Software Considerations in Airborne Systems and Equipment Certification,” RTCA, Inc., 1992.

[83] “DO-254 – Design Assurance Guidelines for Airborne Electronic Hardware,” RTCA, Inc., 2000.

[84] R. Buell, M. Dutton, G. Leggitt, “Replacement Avionics Display Instrument,” U.S. Patent 7,362,240, April 22, 2008.

[85] M. Dutton, “Solid State Avionics Display Instrument,” U.S. Patent 7,446,675, November 4, 2008.

[86] “ATI RadeonTM E4690 delivers leading performance for embedded applications,” [Online]. Available: http://www.amd.com/us/products/ embedded/graphics-processors/Pages/ati-radeon-e4690.aspx. [Accessed: August 1, 2010].

[87] “ALT and Channel One supply industrial rate ATI Radeon E4690 GPUs with 20+ year lifecycle,” [Online]. Available: http://www.altsoftware.com/ company-and-resources/media/past_news/alt-software-and-channelone-E4690.html. [Accessed: August 9, 2010].

[88] “Fujitsu CoralPA, Carmine, Ruby,” [Online]. Available: http://www.altsoftware.com/embedded-products/supplements/fujitsu-coralpa-carmine-ruby.html. [Accessed: August 4, 2010].

[89] “GeForce 7300,” [Online]. Available: http://www.nvidia.com/page/ geforce_7300.html. [Accessed: July 27, 2010].

[90] “PowerVR – Wikipedia, the free encyclopedia,” [Online]. Available: http://en.wikipedia.org /wiki/PowerVR. [Accessed: August 2, 2010].

[91] “PowerVR Technology Overview,” [Online]. Available: http://www.imgtec.com/ factsheets/SDK/PowerVRTechnologyOverview.1.0.2e.External.pdf. [Accessed July 30, 2010].

[92] “IGL 178 – Quantum 3D,” [Online]. Available: http://www.quantum3d.com/ solutions/embedded/igl_178.html. [Accessed: July 28, 2010].

[93] “Designing a safety-certifiable OpenGL software GPU,” [Online]. Available: http://www.vmecritical.com/articles/id/?3702. [Accessed: July 28, 2010].

[94] A. Irturk, S. Mirzaei, R. Kastner, “An Efficient FPGA Implementation of Scalable Matrix Inversion Core using QR Decomposition,” University of California-San Diego Technical Report, CS2009-0938, 2009.

[95] M. Tommiska, "Area-Efficient Implementation of a Fast Square Root Algorithm," *Proceedings of the Third IEEE International Caracas Conference on Devices, Circuits, and Systems*, March 2000.

[96] D. Koch, "Using Tools to Create Tools: CheckPoint Viz - A Case Study," *Proceedings of the International Applied Modeling and Simulation Conference*, pp. 206-211, 2002.

[97] M. Simpson, C. Cox, G. Peterson, G. Sayler, D. Koch, M. Allen, J. Lancaster, J. McCollum, D. Austin, L. Yan, and M. Dutton, "Simulation and Analysis Tools for Stochastic Processes with Applications to the Vibrio fischeri Quorum Sensing System," Poster Presentation, DARPA Bio-COMP PI meeting, Washington, DC Feb 3-5, 2004 .

[98] M. L. Simpson, C. D. Cox, G.D. Peterson, G.S. Sayler, D. Koch, M. Allen, J. Lancaster, J. M. McCollum, D. Austin, L. Yan, and M. Dutton, "Analysis, Modeling, Simulation, and Visualization of the Vibrio fischeri Quorum Sensing System," Platform Presentation, DARPA Bio-COMP PI meeting, Ft. Lauderdale, FL, May 2003.

[99] M. Dutton, D. Keezer, "An FPGA Architecture for Accelerating Graphics Processing and General Purpose Computations," *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines,* 2010.

VITA

Marcus Dutton was born in 1978 in Atlanta, Georgia. He received a B.S.E.E. degree in May 2001 from the University of Tennessee in Knoxville, Tennessee. He has worked for the Display Systems division of L-3 Communications since 2001 in increasing levels of responsibility, currently serving as the Manager of Electrical Engineering. His research focus at L-3 has been primarily in FPGA-based processing to provide solutions for avionics equipment. This research has resulted in two patents, two pending patents, and several publications. While working at L-3, he has continued his studies at Georgia Institute of Technology and received his M.S.E.C.E. degree in May 2004.